LEVERAGING ATTENTION FOCUS FOR EFFECTIVE REINFORCEMENT LEARNING IN COMPLEX DOMAINS

A Thesis
Presented to
The Academic Faculty

by

Luis C. Cobo

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2013

LEVERAGING ATTENTION FOCUS FOR EFFECTIVE REINFORCEMENT LEARNING IN COMPLEX DOMAINS

Approved by:

Professor Charles L. Isbell, Advisor
School of Interactive Computing
Georgia Institute of Technology

Professor Gordon Stüber
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Aaron D. Lanterman, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Andrea L. Thomaz
School of Interactive Computing
Georgia Institute of Technology

Professor Ayanna Howard
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Patricio Vela
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 25 March 2013

A mi prima Lali.

ACKNOWLEDGEMENTS

My most sincere gratitude to Professor Charles Isbell, who believed in me when I needed it most and helped me reorient my research and my career in the direction I wanted to go. I deeply appreciate all the trouble he dealt with so that I only had to care about research, and all his support, advice, and kind words during these years. And the Friday lunches.

I am also indebted to Professor Aaron Lanterman, who has been always available and helpful whenever he was needed. Professor Andrea Thomaz support has been invaluable in the inception and execution of the ideas included in this thesis. It has been a pleasure working with her and her frequent encouragement was a great help to keep up my motivation.

I would like to thank the proposal committee chair, Professor Ayanna Howard, and the rest of the members of the committee, Professor Patricio Vela and Professor Gordon Stüber, for the time they have devoted to my work and the advice, questions and recommendations that have led to a better thesis.

To the former and current members of the pfunk research group, Peng, Kaushik, Jon, Sooraj, Chris, Liam, Arya and Ashley, thank you for being there and for your help reviewing papers and discussing ideas. The same goes for everyone at the RIM lab, Carlos, Ana, Martin, Richard...

I feel blessed that I have had such great bosses when I have been in industry. Without the support of Javier Cardona (CEO of cozybit Inc.), I might not have taken the step to go further and continue my education. I must also thank him for being a great friend, for all the care he took about my well-being while I was his employee and after I left the company, and for allowing me to work on such awesome projects.

I was also lucky in my internships at Google. I owe a lot to my host Alan Skelley, always ready to decrease his productivity so that I could increase mine. Google is a place full of amazing people, and I received a lot of help from many of them. I would like to especially thank the ones whose time I abused the most, Dietmar Ebner, Arnar M. Hrafnkelsson, Brendan McMahan, Gary Sivek and Gary Holt.

Capoeira has played a big role in keeping my sanity and my focus while working on this thesis. I have to thank Brian de Pue (Guile) for introducing me to this art and Mestre Fran and all the Maculelê Atlanta group for the tough training and all the positive energy. Hopefully one day I will play a decent game.

No less important have been the many friends I have made in Atlanta during all this time. Special thanks to José Valenzuela for being so awesome and to Marcia Diehl for the lasagna, the hospitality, and many more things.

I am also very lucky to have so many friends in my hometown, Úbeda, that have not forgotten me and keep gifting me with precious moments every time I visit. Everyone from JAC, "la chirigota" and "los niños de las pipas," specially Antoñojo, Mariajo, Poncho, Dani, Cobo, Carra, Pepe and Antiñolo for hanging out with me more often and longer than they should. I hope we keep seeing each other (more) frequently in the future, and I wish you the best luck in this difficult time our country is going through.

Finally, the most important acknowledgment to my family, to which I owe everything. Thanks to my parents, Luis and Pilar, for supporting me all these years and opening for me the doors to opportunities they never had. Thanks for being such loving parents and for bearing with my sometimes unusual behavior, projects, and travels. Thanks to my sisters, Dina and Puri, and in-laws, Juan Antonio and Jose, for many things, but specially for giving me such a wonderful niece and godchild, Laura, and such wonderful nephews, Ángel, Adrián and Sergio. I hope I soon find a way to be closer to all of you.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ACKNOWLEDGEMENTS . . . . . **iv**

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xi**

SUMMARY . . . . . **xii**

I INTRODUCTION **1**

1.1 Learning from Demonstration 3

1.2 Reinforcement Learning 5

1.3 Our Approach: Attention Focus 6

1.4 Applications 8

1.5 Contributions 10

1.6 Thesis Organization 11

II REINFORCEMENT LEARNING **12**

2.1 Markov Decision Processes 13

2.1.1 Formulation 13

2.1.2 Value function and Q-function 14

2.1.3 Factored representations 15

2.1.4 Types of MDPs 16

2.1.5 POMDPs 17

2.2 Solving MDPs 17

2.2.1 With access to the MDP model 18

2.2.2 With access to a simulator 19

2.2.3 Policy search 20

2.2.4 Value-based RL algorithms 21

2.2.5 Model-based RL algorithms 24

III FUNCTION APPROXIMATION FOR REINFORCEMENT LEARNING . . . 26

3.1 Fitted Q-Learning . . . 26

3.2 LSPI . . . 28

3.3 Limitations of Linear Models . . . 29

3.4 Other Algorithms . . . 30

IV ABSTRACTIONS FOR REINFORCEMENT LEARNING . . . 31

4.1 Hierarchical Abstractions . . . 31

4.2 State-Space Abstraction . . . 33

4.3 Relational Reinforcement Learning . . . 34

V ATTENTION FOCUS . . . 35

5.1 Sources of attention focus information . . . 36

5.1.1 Human demonstrations . . . 37

5.1.2 World models . . . 38

5.2 Application Domains . . . 39

5.3 Relation to Other Function Approximation Algorithms . . . 40

5.4 Relation to Other Abstraction Algorithms . . . 41

VI ATTENTION FOCUS FROM HUMAN DEMONSTRATIONS . 43

6.1 Additional Notation . . . 43

6.2 Abstraction from Demonstration . . . 44

6.2.1 Algorithm . . . 44

6.2.2 Policy invariance . . . 46

6.2.3 Theoretical properties of AfD . . . 48

6.3 Automatic Decomposition and Abstraction from Demonstration . . 49

6.3.1 Algorithm overview . . . 50

6.3.2 Problem decomposition . . . 50

6.3.3 Subtask state space abstraction . . . 55

6.3.4 Policy construction . . . 55

6.3.5 Policy improvement 56
6.4 Combination with Function Approximation 56
6.5 Experimental Evaluation 57
6.5.1 Domains 57
6.5.2 Results on non-hierarchical domains 61
6.5.3 Results on hierarchical domains 65
6.5.4 Function approximation 71
6.6 Discussion 80
6.7 Conclusions 82
VII ATTENTION FOCUS WITH A WORLD MODEL 84
7.1 Modular RL 84
7.2 Object Focused Q-learning 85
7.2.1 Object focused MDPs 86
7.2.2 Algorithm overview 87
7.2.3 Risk threshold and complete algorithm 89
7.3 Benefits of the Arbitration 91
7.4 OF-Q Properties 93
7.4.1 Class-specific policies 93
7.4.2 Risk thresholds 95
7.5 Object Dependencies 96
7.5.1 Approach overview 96
7.5.2 Domains with single-instance classes 97
7.5.3 Domains with many-instances classes 100
7.6 Experimental Evaluation 101
7.6.1 Independent objects 101
7.6.2 Dependent objects 106
7.7 Discussion 112
7.8 Conclusions 113

VIII ATTENTION FOCUS INTERACTIONS 114

8.1 Types of Interactions 114

8.2 Intra-Object Abstraction 116

8.3 Experimental Evaluation 117

8.4 Discussion and generalization 118

IX CONCLUSIONS 120

APPENDIX A — MUTUAL INFORMATION 122

REFERENCES 123

VITA 130

LIST OF TABLES

1 Performance on Pong. 62

2 Frogger domain results with all demonstrations. 63

3 Frogger domain results with best player demonstrations. 63

4 LfD results on Panda domains. 66

5 ADA results on Panda domains. 68

LIST OF FIGURES

1 Learning from demonstration overview. 3

2 Reinforcement learning overview. 5

3 Attention focus from human demonstrations overview. 7

4 Attention focus from world model overview. 8

5 Example MDP losing Markov property on state abstraction. 47

6 Diagram of automatic decomposition and abstraction from demonstration. 52

7 Screen capture of the Frogger domain. 58

8 Capture of the RainbowPanda domain. 60

9 Performance of AfD vs. RL in Frogger. 64

10 Policy size per steps on AfD and RL. 65

11 Sarsa results on simplified PandaSequential. 67

12 Fitted Q-learning on Frogger. 72

13 LSPI on Frogger. 74

14 Fitted Q-learning on Panda. 76

15 LSPI on Panda. 79

16 Arbitration problems. 92

17 OF-Q domains. 101

18 OF-Q results. 103

19 OF-Q results for different initial thresholds. 104

20 One-Knob domain representation. 107

21 One-Knob domain results. 109

22 Create-door domain results. 109

23 Direct-reward domain results 110

24 Two-Knob domain results. 110

25 Combination domain results. 111

26 Results combining AfD and OF-Q. 117

SUMMARY

One of the hardest challenges in the field of machine learning is to build agents, such as robotic assistants in homes and hospitals, that can autonomously learn new tasks that they were not pre-programmed to tackle, without the intervention of an engineer. Reinforcement learning (RL) and learning from demonstration (LfD) are popular approaches for task learning, but they are often ineffective in high-dimensional domains unless provided with either a great deal of problem-specific domain information or a carefully crafted representation of the state and dynamics of the world. Unfortunately, autonomous agents trying to learn new tasks usually do not have access to such domain information nor to an appropriate representation.

We demonstrate that algorithms that focus, at each moment, on the relevant features of the state space can achieve significant speed-ups over previous reinforcement learning algorithms with respect to the number of state features in complex domains. To do so, we introduce and evaluate a family of *attention focus* algorithms. We show that these algorithms can reduce the dimensionality of complex domains, creating a compact representation of the state space with which satisficing policies can be learned efficiently. Our approach obtains exponential speed-ups with respect to the number of features considered when compared with table-based learning algorithms and polynomial speed-ups when compared with state-of-the-art function approximation RL algorithms such as LSPI or fitted Q-learning.

Our attention focus algorithms are divided in two classes, depending on the source of the focus information they require. *Attention focus from human demonstrations* infers the features to focus on from a set of demonstrations from human teachers performing the task the agent must learn. We introduce two algorithms within this

class. The first one, *abstraction from demonstration* (AfD), identifies features that can be safely ignored in the whole state space and builds a state-space abstraction where a satisficing policy can be learned efficiently. The second, *automatic decomposition and abstraction from demonstration*, goes one step further, using the demonstrations to identify a set of subtasks and to find an appropriate abstraction for each subtask found.

The other class of algorithms we present, *attention focus with a world model*, does not require a set of human demonstrations. Instead, it extracts the attention focus information from an object-based model of the world together with the agent experience in performing the task. Within this class, we introduce *object-focused Q-learning* (OF-Q), at first with an assumption of object independence that is later removed to support domains where objects interact with each other. Finally, we show that both sources of focus information can be combined for further speed-ups.

CHAPTER I

INTRODUCTION

One of the hardest challenges in the field of machine learning is to build agents that can autonomously learn new tasks that they were not pre-programmed to tackle, without intervention of an engineer. Agents such as robotic assistants in homes and hospitals or non-player characters in videogames may be required to perform tasks beyond the imagination of the agent designers, and different end users may require different behaviors. This makes it impractical to manually engineer a policy for all the possible tasks the agent may have to perform. Typically, agents see these tasks as sequential decision problems in which the agent perceives the state of the world, decides and executes the best action to take in this state, and then the agent perceives the new state of the world. The aim of learning agents is to find a *policy* that maps each possible state of the world to the optimal action to take in that situation.

Reinforcement learning (RL) and learning from demonstration (LfD) are popular families of algorithms for solving sequential decision problems, but they are often ineffective in high-dimensional domains unless provided with either a great deal of problem-specific domain information or a carefully crafted representation of the state and dynamics of the world. Unfortunately, autonomous agents trying to learn new tasks usually do not have access to such domain information nor to an appropriate representation.

We demonstrate that demonstrate that algorithms that focus, at each moment, on the relevant features of the state space can achieve significant speed-ups over previous reinforcement learning algorithms with respect to the number of state features in complex domains. To do so, we introduce

and evaluate a family of *attention focus* algorithms. We target real-world tasks that can be performed by humans, because these are the domains that may benefit from attention focus mechanisms similar to the ones our brain uses to cope with the high dimensionality of the physical world. Our work we focuses on efficiently obtaining satisficing or near-optimal policies in domains where algorithms that aim for strict optimality do not converge in any reasonable amount of time. Our algorithms reduce the need for manual feature engineering, so they can be useful for multipurpose autonomous agents that must learn tasks that are not known in advance.

Most research on machine learning assumes a suitable representation (or state space) for the problem to be solved, but finding this representation is often the hardest part of solving real-world tasks. Usually, this representation is found by machine learning practitioners manually, with a combination of experience, domain insight, and intuitive knowledge [19]. Our work opens the door to more systematic approaches to deriving appropriate representations for a problem automatically.

The algorithms we propose are extensions of RL that draw some elements from LfD techniques. We continue this chapter by briefly introducing these two approaches and then discussing some of the problems of reinforcement learning and how they might be mitigated. To conclude our introduction, we describe applications of our work and summarize our contributions.

1.1 *Learning from Demonstration*

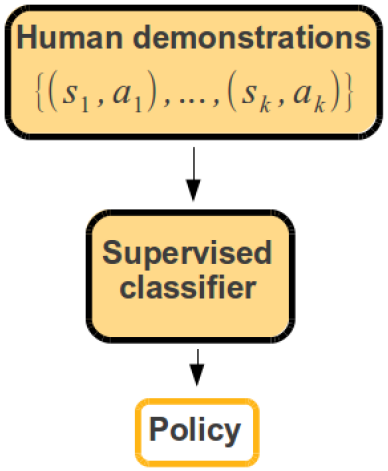


Figure 1: Learning from demonstration overview.

Learning from demonstration (LfD) [4] is a family of algorithms that learn a policy for a sequential decision problem by reducing it to a supervised learning problem. For this, it is first necessary to acquire a set of demonstrations, usually from human teachers performing the task. These demonstrations are represented as state-action tuples that associate an action taken in the demonstrations with a specific state of the world. The demonstrations can be provided, for example, by human teachers tele-operating a robot. Other types of demonstrations are possible, such as a human teacher executing the task (*e.g.,* kicking a ball) herself, but that brings the additional problem of mapping the teacher's view of the world to the perspective of the learning agent. Our work assumes that demonstrations are already adapted to the agent perspective. Once we have these state-action demonstrations, we can use any appropriate supervised classification algorithm to derive a policy that will associate each state to its optimal action, as shown in Figure 1.

In principle, LfD could be effective in large state spaces because it focuses on the interesting regions of the state space, which are the same regions that are shown in the demonstrations. However, these approaches often do not work well unless either

a detailed model of the problem is available or the learned policy is just used as a starting point for other learning techniques [2, 12]. The number of samples required to derive a good policy may be very large, and obtaining human demonstrations is costly and time-consuming. Besides the cost, obtaining demonstrations is difficult because humans get bored and distracted easily when performing a task repeatedly. Further, training and testing distributions for the classification algorithm may not match when the agent has not perfectly mimicked the teacher. Even if a classification algorithm predicts correctly the action that the human teacher would have taken for 99% of the queries (typically a good result for supervised learning), this may not be good enough given the sequential nature of task learning. The 1% error can accumulate in consecutive time steps and lead the agent to an unknown part of the state space, where the performance of the classification algorithm will be very poor because there are no demonstrations from that region. Thus, small errors can produce large differences in the distribution of states that the agent will encounter and dramatically decrease performance.

A significant branch of LfD focuses on combining demonstrations with traditional RL methods, including using demonstrations to guide exploration [68, 38] or learn a reward function [1]. In our work, we use LfD to build state-space abstractions and task decompositions that are then combined with RL algorithms. Previous work using LfD to induce task decompositions [53, 84] required a dynamic Bayesian network (DBN) model of the environment, while our methods are model free.

Another branch of LfD uses demonstration data for learning plans, *e.g.,* learning pre and post conditions; however, this work typically assumes that additional information, such as annotations, is provided to the algorithm. We limit our work to cases where only demonstrations (at most) are available, reflecting our focus on agents that can learn autonomously from non-experts.

1.2 Reinforcement Learning

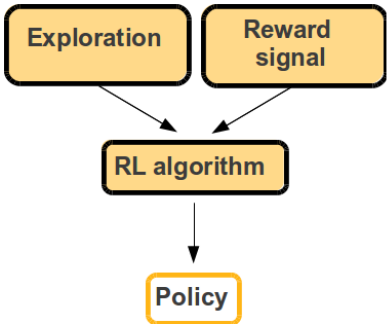


Figure 2: Reinforcement learning overview.

Reinforcement learning (RL) algorithms [72] learn a policy by letting the agent explore the effects of different actions in different situations while trying to maximize a sparse reward signal. This is fundamentally different from supervised learning, where each sample is independent from the others and has a specific label associated with it. RL has to address the problem of *temporal credit assignment*. Imagine, for example, a game of chess, where the only reward is obtained at the end of the game, with values 1, 0, or -1 for win, draw, or loss, respectively. The agent may lose a game after having been playing optimally for the last n moves because of a bad choice $n+1$ moves ago. The challenge in RL is to identify which actions from a sequence of state-actions are responsible for a given outcome, so that in our chess example the agent learns which move was a bad choice and the subsequent optimal play is not penalized by the negative reward.

RL has been successfully applied to a variety of scenarios [77, 55]; however, RL tends to not scale well to high-dimensional state spaces because of the *curse of dimensionality*. As the number of features that compose the state space grows linearly, the size of the state space grows exponentially, and so does the time required to converge.

This problem has been hitherto addressed by two different but related approaches: manual engineering of the features and function approximation. In manual feature

engineering, an expert analyzes the task to be learned and, starting from a large set of low-level features, composes a small set of high-level features with which the agent can learn a compact and good policy for the task. This approach, however, defeats the purpose of learning algorithms, since it shifts the engineering task from designing a policy to designing the feature space in which the policy will be learned. The approach can be useful if the latter task is easier than the former; however, we cannot talk of true autonomous learning agents if a significant amount of manual feature engineering is needed for each task to be learned.

Function approximation, the other approach to deal with the curse of dimensionality in RL, offers significant speed-ups with respect to learning on a flat tabular representation of the state space. Unfortunately, function approximation often requires manual feature engineering, as the hypothesis space of the approximation algorithm (*e.g.,* linear combinations of the features) may not be appropriate if used with low-level features.

1.3 Our Approach: Attention Focus

Our approach is to devise algorithms that can automatically focus on a small set of relevant features at each point in time. This set often changes during the performance of a task. We call this family of algorithms *attention focus algorithms*.

Attention focus cannot help in domains where every feature is necessary at each moment. It will not help, for example, in a task with a large number of independent binary features if the optimal action at each moment depends on the checksum of all the features. However, real-world tasks, especially tasks performed by humans, usually have a structure that can be leveraged. Studies on human psychology point in this direction. It is estimated that our brains can simultaneously receive up to 11 million pieces of information, but we can be consciously aware of at most 40 of these [81]. Further, adults can only hold a maximum of three to five meaningful items

or "chunks" in their working memory [16], the short-term memory used for cognitive tasks. This suggest that humans perform even the most complex tasks by focusing only on a small number of features at each moment. Our work shows that the same approach can be applied to RL algorithms.

Function approximation for RL can also reduce the feature set either by implicit feature selection or explicit regularization, but these approaches aim at finding an optimal policy whose representation may need a large number of features. In our attention focus approach, however, we sacrifice optimality and aim instead for *satisficing* policies that can be expressed paying attention only to a small set of features at a time and that can therefore converge quickly. The term "satisficing" refers to a policy that is useful but may not be strictly optimal. This trade-off is beneficial in problems that, because of their complexity, prevent traditional algorithms to converge in any reasonable amount of time.

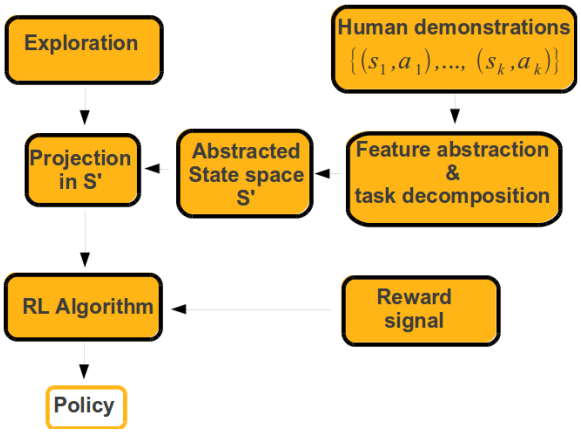


Figure 3: Attention focus from human demonstrations overview.

To determine appropriate features to focus on, we use two different approaches. The first approach, depicted in Figure 3, is to infer which features humans pay attention to when performing the task, extracting this information from a set of humans demonstrations of the task the agent must learn. Chapter 6 shows that from a small set of these demonstrations we can derive both a state-space abstraction and a task

decomposition that we can use to speed up learning. In our experiments, we observed that to find a good state representation that could speed up significantly RL, it sufficed to have a set of demonstrations that was more than an order of magnitude too small for LfD algorithms to find a good policy.

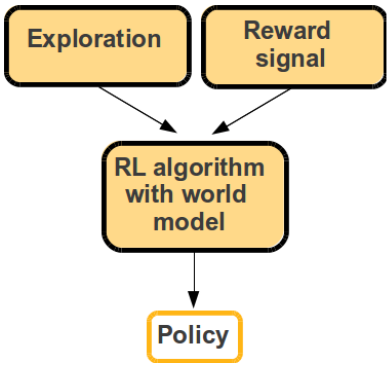


Figure 4: Attention focus from world model overview.

The second approach, which we detail in Chapter 7, is to derive the attention focus from general world models and from the experience of the agent performing the particular task. These algorithms, depicted in Figure 4, do not require demonstrations. Instead, they use an RL algorithm that includes a model of attention focus that helps break down the complexity of the problem. Our attention focus model represents the state space as a collection of nearly independent objects, organized into object classes, that can interact with the learning agent or among themselves. Using our models, a learning agent can explore the state space to determine the correct attention focus for a particular task, requiring only a minimal amount of task-specific domain information. This minimal need of domain information differentiates our approach from earlier relational RL algorithms that we review in Section 4.3.

1.4 Applications

Attention focus extends the class of problems that we can feasibly solve with reinforcement learning. We focus on domains where sacrificing strict optimality is necessary

because of their high complexity, which can be NP-hard [3]. RL tends to work well only when the state of the world is described by just a handful of features, but this is usually not the case in realistic tasks. By identifying the relevant features to pay attention to, we can attack bigger domains and find policies that generalize better. Attention focus algorithms could, for example, enable a hypothetical omniscient soccer-playing robot to learn that the color of the T-shirt of the opponent and how many spectators are watching a match are irrelevant for its policy. This allows a more efficient reuse of experience and provides a degree of transfer learning.

The same benefit could be obtained if an engineer manually specifies which features are the relevant ones for each possible task. However, requiring an engineer every time an agent needs to learn a new task may be unrealistic or economically impractical. Attention focus helps autonomous agents to learn new tasks after deployment to match the needs and preferences of end users.

In our experimental setup, we have used several videogames because they are convenient domains to simulate and obtain human demonstrations for. They also make it straightforward to match the human's representation and the agent's internal representation of the problem. However, our work applies to any kind of autonomous agent that needs to learn after deployment, for example, manufacturing robots in industry or robot assistants aiding at domestic tasks, automatizing procedures in hospitals, or providing assistance to elderly or disabled individuals. These multipurpose agents typically have a wide range of sensors that can provide a high-dimensional signal about the world. In general, this high-dimensional signal is too complex for direct learning. Attention focus from demonstration is likely not useful on top of this raw signal either, but these signals can be preprocessed using unsupervised feature learning techniques such as deep learning [44] to provide a set of features that are still too large for direct RL, but are suitable for attention focus algorithms. Alternatively, the agents can use off-the-shelf vision algorithms [8] to provide a signal that could be

used by our object-based attention focus models.

1.5 Contributions

This thesis introduces three algorithms, AfD, ADA and OF-Q [15, 13, 14], that leverage attention focus to speed up previous RL algorithms and extend the class of domains that can be effectively solved with RL techniques. At the same time, our algorithms reduce the need for manual feature engineering, thus making the learning process more automatic, which is key for autonomous and multipurpose learning agents. Our work can be summarized by the following contributions to the field of machine learning:

- Methods to derive the correct dynamic attention focus for a task from human demonstrations and from the agent's own experience in performing the task.
- A family of attention focus algorithms for autonomous agents that offers significant speed-ups over state-of-the-art algorithms with respect to the number of state features.
- An experimental and theoretical analysis of attention focus methods that specifies the speed-ups that these algorithms offer.

Humans leverage selective attention, their form of attention focus, extensively to perform every day tasks [81, 16]. To our knowledge, this idea has not previously been applied to algorithms for learning sequential decision tasks. We consider the introduction of this idea in the field as an additional, broader contribution, and we hope that our work will inspire other researchers to use attention focus in innovative ways in their future work.

1.6 Thesis Organization

Chapter 2 provides a general introduction to reinforcement learning, with special emphasis on the aspects that are key to our work. Chapters 3 and 4 review existing work on function approximation techniques and abstractions for reinforcement learning, because our attention focus approach is related to these two families of techniques.

Chapter 5 offers a general overview of attention focus algorithms, with the two following chapters giving a thorough description and experimental evaluation of our algorithms for the two attention focus approaches that we introduce. Finally, Chapter 8 discusses how both approaches to attention focus can work together, and Chapter 9 summarizes the overall conclusions of our thesis.

CHAPTER II

REINFORCEMENT LEARNING

Machine learning algorithms are usually divided into different families depending on the problem settings they apply to: supervised learning, unsupervised learning, semi-supervised learning, dimensionality reduction, and reinforcement learning. Supervised algorithms learn to predict the label of previously unseen instances by generalizing from a set of training instances with their appropriate labels. The instances could be, for example, a set of images of handwritten digits with the labels indicating which digit they represent. Problems with discrete labels are referred to as classification problems, and problems with continuous labels are called regression problems. Unsupervised algorithms are different in that they aim at finding structure in a set of instances in the absence of labels. Such algorithms can, for example, organize digit images into clusters of images that represent the same number, or at least, a specific style of writing the number. Semi-supervised algorithms combine elements from the two previous algorithms. Dimensionality reduction transforms the input instances so that they are more suitable for use with the previous algorithms.

Reinforcement learning [72] (RL), the field our research extends, is fundamentally different from these other machine learning areas. While all the previously mentioned approaches are concerned with individual instances, RL works with sequences of state-actions of varying or even infinite length. Each state-action is roughly similar to an instance, typically a fixed-length set of features. Additionally, while other algorithms are usually provided with a fixed training set, RL algorithms perform active exploration and have influence in the distribution of the training samples they receive. RL solves problems where an agent must take a sequence of actions in a changing

environment to achieve a goal, with the environment changing at least in part in response to the actions taken by the agent. In RL, there is no information on whether a specific action is good or bad; otherwise, we could simply frame the problem as a supervised learning problem with the states as instances and the best actions to take as labels. Instead, the algorithm leverages a sparse reward signal, often received only at the end of the task, that encodes how well the agent performed. The aim of RL is to maximize this reward signal by solving the *temporal credit assignment* problem, this is, given a sequence of states, actions, and rewards, to discover which actions had a positive or negative impact in the received reward and which are thus the best actions to take in each possible state.

2.1 Markov Decision Processes

Reinforcement learning algorithms solve sequential decision processes, in which an agent must make a series of decisions depending on the state of the environment. The number of possible sequences of state-actions can be infinite, even if there are a small number of actions and states. Therefore, if the dynamics of the world depend on the whole sequence of events, learning is unfeasible. For this reason, RL is mainly concerned with problems where the next state of the world depends only on the current state of the world and the action that the agent takes, independent of the previous history of state-actions. This is called the Markov property, and sequential decision processes that have this property are called Markov Decision Processes (MDPs).

2.1.1 Formulation

MDPs are formalized as a tuple

$$M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)\,,$$

with the following elements:

$\mathcal{S}$ is the *state space*, a set of possible states.

$\mathcal{A}$ is the *action space*, a finite set of actions that can be taken by the agent.

$\mathcal{P} = \mathcal{P}^a_{ss'} = Pr\,(s'|s,a)$ is the *transition function*, *i.e.,* the probability of transitioning to state $s' \in \mathcal{S}$ when taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. Because of the Markov property, the current state and action are enough to determine this transition probability.

$\mathcal{R} = \mathcal{R}^a_s = r(s,a)$ is the *reward function*, which determines the immediate reward obtained when taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. It can be stochastic.

γ is the *discount factor* that encodes an implicit preference between immediate rewards or larger rewards in the future, where $0 < \gamma \leq 1$.

An MDP episode is organized sequentially as a series of steps $t = 0, 1, 2, \ldots$, with step $t = 0$ being the initial step. At each time step t, the agent perceives the state of the world $s_t \in \mathcal{S}$, takes an action $a_t \in \mathcal{A}$, and receives a reward $r_t = r(s_t, a_t)$. The total reward for the agent in these settings is the *sum of discounted rewards*

$$\sum_t \gamma^t r_t = \sum_t \gamma^t \mathcal{R}^{a_t}_{s_t}.$$

A *policy* $\pi : S \to A$ determines which action $a \in \mathcal{A}$ to take in each state $s \in \mathcal{S}$. The objective of reinforcement learning is to find an *optimal policy* π^*, *i.e.,* a policy that obtains the highest expected sum of discounted rewards.

2.1.2 Value function and Q-function

Given an MDP $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ and a policy π, we can define the state-value of a state $s \in \mathcal{S}$ with respect to policy π as

$$V^\pi(s) = \mathcal{R}^{\pi(s)}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^{\pi(s)}_{ss'} V^\pi(s'),$$

the expected sum of discounted rewards when starting at state s and following policy π. $V^*(s) = V^{\pi^*}(s)$ is the state-value of state s when following an optimal policy π^*. This value is also referred to simply as the value of state s, or $V(s)$.

As we will see later, sometimes it is convenient to work with values for state-action pairs $(s, a), s \in \mathcal{S}, a \in \mathcal{A}$. These are called Q-values, and are formulated as

$$Q^\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a Q^\pi(s', \pi(s')), \tag{1}$$

the discounted reward obtained when taking action a in state s and subsequently following policy π. Q-values can also be defined in terms of state values, and vice versa:

$$Q^\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a V^\pi(s'), \tag{2}$$

$$V^\pi(s) = \max_a Q^\pi(s, a). \tag{3}$$

$Q(s, a) = Q^*(s, a) = Q^{\pi^*}(s, a)$ are analogous to their state-value function counterparts. Note that the state-values and Q-values are always defined with respect to a given policy that is not necessarily optimal.

2.1.3 Factored representations

The state space $\mathcal{S}$ is often represented, not as a set of unrelated states, but as a *factored representation*, *i.e.,* , a cross product of n features,

$$S = F_1 \times \cdots \times F_n,$$

where each feature F_i has an independent range of values. Each possible state $s \in \mathcal{S}$ is then defined by the values of these features:

$$s = (f_i, \ldots, f_n), f_i \in F_i.$$

Factored representations are practical because they enable generalization, *i.e.,* extending what is learned in one state to similar states. These representations also

allow the application of other machine learning algorithms, such as dimensionality reduction or unsupervised learning, to improve learning efficiency. Furthermore, factored representations more naturally represent a world state that is almost always the result of a combination of different elements. For these reasons, our work focuses on such representations.

In a factored representation, an additional feature multiplies the number of possible states. This means that a linear increase in the number of features considered leads to an exponential increase in the number of states in the domains, which consequently leads to an exponential increase in the time required to learn a policy. Our attention focus algorithms leverage this fact to provide significant learning speed-ups.

2.1.4 Types of MDPs

MDPs can be classified in several ways, depending on whether the reward is discounted ($\gamma < 1$) or not, whether we are considering a finite (bounded t) or infinite horizon, whether the task is episodic (*i.e.,* there are final absorbing states) or not, and whether the transition and reward functions are stochastic.

In our work, we want agents to learn every day tasks that can be performed by humans. These tasks are in general episodic and stochastic, and they do not have a fixed length. In these tasks, there are often end success and failure states (whether the task was accomplished or not), and it is preferable to reach a positive final state as soon as possible. For this reason, our experimental domains are stochastic, episodic, and have an infinite horizon. To encode the urgency of reaching a positive final state, we use either a discount factor $\gamma < 1$ or a per-step cost (negative reward) along with no discount ($\gamma = 1$).

2.1.5 POMDPs

An important extension to MDPs are partially observable MDPs (POMDPs). POMDPs are MDPs where the current state of the world is not known; instead, the agent receives a series of observations. The agent must learn the correlation between the observations and the true states of the underlying MDP and must keep a belief distribution over all the possible states over time. This requires considering the total history of observations, because even though the underlying MDP keeps the Markov property, the sequence of observations is, in general, not Markovian.

Our work assumes that the domains are fully observable, but some of our algorithms abstractions can induce state aliasing, where some states are clustered together. Our algorithms are designed so that this aliasing does not impede learning, as only irrelevant components of the state space are ignored and the partial observability can be folded into the stochasticity of the domain.

2.2 Solving MDPs

The objective of reinforcement learning is to find the optimal policy for an MDP M, without having access to the transition model $\mathcal{P}$ nor the reward model $\mathcal{R}$. Knowledge of the state space $\mathcal{S}$, action space $\mathcal{A}$, and discount factor γ is always assumed. RL algorithms rely on their interactions with the environment, governed by the unknown $\mathcal{P}$ and $\mathcal{R}$, to find the optimal policy. Our work also assumes that the model can only be known through interaction with the environment, but for completeness, we briefly discuss techniques that can be used when we have access to either the model or a simulator of it. Then, we present two different types of model-free RL methods: policy search and value-based. Finally, we introduce model-based RL methods, which do not have prior knowledge of the MDP transition and reward models but try to infer them from experience.

2.2.1 With access to the MDP model

If the transition and reward functions are known, the optimal policy π^* can be found using dynamic programming algorithms such as *policy iteration* and *value iteration*. These methods are particularly important because they are the foundation of the reinforcement learning algorithms that we will discuss later.

2.2.1.1 Policy iteration

Policy iteration [72] is divided in two phases. At the beginning, it uses an arbitrary policy $\pi(s)$ and assigns arbitrary values $V(s)$ to each state s. The first phase is *policy evaluation*, where the algorithm computes the true state-values corresponding to the current policy $\pi(s)$. For this, the algorithm sweeps through each state $s \in \mathcal{S}$, performing the update

$$V(s) \leftarrow \mathcal{R}_s^{\pi(s)} + \gamma \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} V(s').$$

This policy evaluation sweeps over the entire state space S several times until the state-values converge or the maximum update to a state-value in the last sweep is smaller than a threshold ϵ.

Once policy evaluation is completed, the policy is updated to maximize the expected reward with respect to the current value-function. In this second phase, called *policy improvement*, only one sweep through the entire state space is necessary, updating the policy for each state according to the expression

$$\pi(s) = \arg\max_a \mathcal{R}_{ss'}^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a V(s').$$

If the policy does not change for any state, then $\pi(s) = \pi^*(s)$ and the algorithm exits. Otherwise, it goes back to the policy evaluation step.

Policy iteration always converges to the optimal policy, sometimes in a small number of iterations [72]. However, besides requiring knowledge of $\mathcal{P}$ and $\mathcal{R}$, the sweeps over the entire state space can be unfeasible in realistic, high-dimensional

tasks. Methods based on policy iteration are also called *actor-critic methods*, with the actor referring to the policy improvement step, which controls the agent behavior, and the critic referring to the policy evaluation step.

2.2.1.2 Value iteration

Value iteration combines the two phases of policy iteration into one, performing repeated sweeps over the entire state space, updating each state value with

$$V(s) \leftarrow \max_a \mathcal{R}_s^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a V(s'),$$

which is known as the Bellman equation [72] .

The algorithm keeps sweeping the entire state space until the value function converges or the largest update to one state-value is smaller than a given threshold ϵ. Then the state values will be the optimal state values V^*, and the optimal policy will be

$$\pi^*(s) = \arg\max_a \mathcal{R}_s^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a V^*(s'). \tag{4}$$

Like policy iteration, value iteration is guaranteed to converge to the optimal policy, but it is usually impractical for realistic tasks.

2.2.2 With access to a simulator

Another scenario assumes no knowledge of the transition and reward models, but instead assumes access to a simulator that the agent can use to test the effect of different actions in any given state. In this case, Monte-Carlo tree search methods [26] (MCTS) can be used, along with extensions such as upper confidence trees (UCT). These methods have shown great success in difficult domains such as the strategy board game Go, even though the simulators are only approximate (as the exact strategy of the opponent is not known).

While these methods are important both theoretically and in applications, they

cannot be used in the case of an agent that must interact with an unknown environment in order to learn a policy. In fact, these algorithms are not concerned with learning a policy for the entire state space, but just with deciding which is likely to be the more convenient action in a specific state. Our work assumes no access to a simulator of the environment; and therefore, we must resort to reinforcement learning methods.

2.2.3 Policy search

Policy search algorithms [56, 74] are reinforcement learning methods that aim at finding the optimal policy without direct knowledge of the transition and reward models. Because RL algorithms do not have access to samples of correct state-action pairs to learn from, the policy evaluation must be done through rollouts, *i.e.,* using the policy in the environment as long as necessary to obtain an accurate estimate of the expected discounted reward obtained when using the policy. Unfortunately, these rollouts can be expensive, as episodes can be long, especially when following a poor policy. Additionally, the space of possible policies Π is huge: $|\Pi| = |S|^{|A|}$, so evaluating all possible policies is often unfeasible. However, the aggressive state-space abstraction that attention focus algorithms provide can sometimes reduce the state space enough so that direct policy search is an effective option.

Policy search is more practical if we define a restricted class of policies to consider, particularly if that restricted class is parametrized in a way that allows to use the policy gradient to speed up search. Unfortunately, these policy classes are task-specific and have to be defined manually, which is not a trivial task. It requires a great deal of expertise and domain knowledge to define a compact policy class that still contains an optimal or near-optimal policy. While these options can be valuable for solving specific problems, they fall out of the scope of our work which is to make reinforcement learning more automatic by reducing the need of manual feature (or

policy space) engineering.

2.2.4 Value-based RL algorithms

Model-free value-based algorithms are some of the best known and most widely used RL methods. Value learning methods are similar to value iteration, but without access to the transition and reward functions, they have to base their estimates of state values on their experience interacting with the world. Also, without access to the transition and reward models, transition probabilities cannot be used to derive a policy using (4) as in the case of value iteration. For this reason, value learning algorithms typically learn Q-values, defined in (1) and (2). Once the optimal Q-values are known, the optimal policy is

$$\pi^*(s) = \arg\max_a Q^*(s, a). \tag{5}$$

2.2.4.1 Exploration vs. exploitation

During learning, value-based RL approaches usually pick the action with the current highest estimated Q-value. At the start of the learning process, the Q-values are initialized arbitrarily, so the initial policy is also arbitrary. However, as learning progresses, these algorithms will tend to explore the most promising (*i.e.,* with higher Q-value estimates) regions of the state space. This is desirable because it speeds up learning in the regions of the state space that are most relevant; however, if the algorithms always try to carry out the action with the highest estimated Q-value, they will get stuck in suboptimal policies that rely heavily on anecdotal evidence gathered in the early stages of learning. This problem is known as the *exploration-exploitation trade-off* [72] and it is crucial for reinforcement learning.

The most common approach to address this problem is to use an ϵ-greedy policy, *i.e.,* a policy that takes a random exploratory action with probability ϵ, and takes the action with higher Q-value otherwise. In the exploratory actions, any action can

be taken with equal probability, so the probability of picking any non-optimal action is $\frac{\epsilon}{|\mathcal{A}|}$, while the probability of taking the optimal action, assuming there is only one, is $1 - \frac{\epsilon(|\mathcal{A}|-1)}{|\mathcal{A}|}$. Usually, learning starts with a high value of ϵ that is decreased over time as the confidence in the Q-value estimates grows.

An alternative to an ϵ-greedy policy is a different kind of softmax policy in which the probability of picking each action depends on their Q-value estimate, used as a weight for a Boltzmann, *i.e.,* Gibbs, distribution

$$Pr(\pi(s) = a) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_{a' \in \mathcal{A}} e^{\frac{Q(s,a')}{\tau}}},$$

where τ is a design parameter, sometimes called *temperature*, similar to ϵ, that is also decreased over time.

2.2.4.2 Monte-Carlo methods

Monte-Carlo methods, which must not be confused with the Monte-Carlo tree search algorithm described in Section 2.2.2, are probably the simplest value-based RL algorithms. Starting with arbitrary Q-values, these algorithms keep a list of the discounted rewards obtained when taking each action a from each state s. At each moment, the Q-value estimate is the average of the discounted rewards obtained for that Q-value so far. When these Q-values estimates are used with an ϵ-greedy policy, the policy is guaranteed to converge to the optimal ϵ-greedy policy [72].

Monte Carlo does not bootstrap, this is, it does not use the Q-value estimates of some states to update the Q-values of others. This makes it slower than bootstraping algorithms such as Q-learning or Sarsa [72], but it might be appropriate when used with some state abstractions that interfere with the boostrapping process.

2.2.4.3 Q-learning

Q-learning is, like Sarsa, an example of a time-differences (TD) method. Instead of waiting until a reward is obtained to update the Q-value estimates, TD methods

rely on the expectations of a future reward, derived from the state-values, to update Q-values.

Q-learning starts, as usual, with arbitrary Q-values and a softmax policy based on (5). Every time action a is taken from state s, the agent obtains reward r, the environment transitions to state s', and the corresponding Q-value is updated according to

$$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') \right), \tag{6}$$

where r is the immediate reward obtained, and α a parameter called the *learning rate* or *step size*.

If we use parameter α_t for time step t and assume each state-action is visited infinitely often, the Q-value estimates converge to the true Q-values of the optimal policy as long as $\sum_t \alpha_t = \infty$, $\sum_t \alpha_t^2 < \infty$ [79].

Q-learning is an *off-policy* algorithm, which means it can learn the Q-values corresponding to the optimal policy while following a different *exploration* policy (*e.g.,* an ϵ-greedy policy), as long as all state-actions are visited enough times. This is an important feature that will be used to speed up learning in some attention focus algorithms. Recently, Speedy Q-learning [5], a variant with faster convergence, has been introduced.

2.2.4.4 Sarsa

Another popular bootstrapping algorithm is Sarsa, which stands for *state action reward state action*. It is similar to Q-learning, but instead of updating Q-values with the highest Q-value of the next state, it performs the update using the Q-value that corresponds to the action a' that is actually taken in the next state s':

$$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha \left(r + \gamma Q(s',a') \right). \tag{7}$$

Obviously, the action taken in s' will depend on the exploration policy that the algorithm is following. This makes Sarsa an on-policy algorithm, since it can only

improve the same policy it is using for exploration.

Sarsa is interesting because it makes it easy to implement *eligibility traces*, in the extended algorithm called Sarsa(λ). When using eligibility traces, Q-values are updated considering all the time differences until the end of the episode. If, at time step t, the agent executes action a_t in state s_t and obtains reward r_t,

$$\begin{aligned} Q(s_t, a_t) \leftarrow Q(s_t, a_t) &+ \alpha \left(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right) \\ &+ \alpha\gamma\lambda \left(r_{t+1} + \gamma Q(s_{t+2}, a_{t+2}) - Q(s_{t+1}, a_{t+1})\right) \\ &+ \alpha\gamma^2\lambda^2 \left(r_{t+2} + \gamma Q(s_{t+3}, a_{t+3}) - Q(s_{t+2}, a_{t+2})\right) \\ &+ \cdots \end{aligned} \tag{8}$$

Parameter λ determines how much later TD updates are relevant with respect to earlier ones; how to set it optimally is an open research question [42]. In domains with a lot of steps from the initial to the end states, using traces can significantly accelerate convergence, since reward information typically obtained in the end states reaches the start states much more efficiently. Although there exist some adaptations of Q-learning that use eligibility traces, they tend to not work as well because traces must be cut each time an exploratory step is taken.

2.2.5 Model-based RL algorithms

Model-based RL algorithms estimate the MDP transition and reward models from experience interacting with the environment. Then they compute optimal policies according with the estimated models, using, for example, policy or value iteration. These approaches offer a trade-off with respect to model-free methods. Model-based methods make better use of the experience obtained from the environment, so they require, at least theoretically, fewer interactions with the environment. However, model-based algorithms are significantly more computationally expensive, due to having to derive a policy from the estimated model several times during the learning process. R-max [9] is a representative model-based algorithm with good convergence

bounds. V-max [62], a similar algorithm with identical bounds but better empirical performance, has been recently proposed.

Our work focuses on model-free algorithms because we use videogame domains, where it is cheap to obtain samples from the environment. However, we believe that model-based algorithms can also benefit from attention focus algorithms, because attention focus can significantly reduce the state space and therefore speed up the policy computation step and improve generalization in model-based approaches.

CHAPTER III

FUNCTION APPROXIMATION FOR REINFORCEMENT LEARNING

Function approximation (FA) makes it possible to successfully scale up reinforcement learning (RL) for use in complex real-world domains. However, FA often requires careful and time consuming feature engineering: depending on the chosen state representation, an FA algorithm can be successful or useless in a given domain.

We combine our attention focus algorithms with function approximation because real-world problems may be too complicated for a traditional tabular representation even after an attention focus abstraction is applied. Section 6.4 will show that attention focus algorithms combined with previous FA algorithms can attack larger domains than either of them alone. Seen from another perspective, attention focus can help automate the feature engineering required by FA algorithms, which is crucial for autonomous learning agents.

In our experiments, we use two of the most widely used FA algorithms: fitted Q-learning and LSPI. The following sections will briefly introduce them and detail the specifics of our implementations. Later, we will comment on the limitations of linear models, a class that includes these two algorithms, and finally we will overview other FA algorithms and explain why they are less suitable for our purposes.

3.1 Fitted Q-Learning

Fitted Q-learning [75] is a simple function approximation algorithm that is widely used and works well in practice. In its batch form, it relies in a generic regressor, initialized arbitrarily, that is accessed through a prediction interface and a regression

interface. The prediction interface predicts the Q-value for any state-action pair; the regression interface takes a training set of state-action pairs associated with the Q-value they should have and outputs a regressor that fits those samples. The samples are generated using the predictions $\tilde{Q}(s,a)$ of a previous regressor and a set of transitions (s,a,r,s') observed by the agent interacting with the environment:

$$(s, a, r + \gamma \max_{a' \in \mathcal{A}} \tilde{Q}(s', a')). \tag{9}$$

Fitted Q-learning updates the regressor, uses it to re-estimate the expected Q-values, and refreshes the regressor until convergence. The specific convergence guarantees that can be made depend on the nature of the regressor used.

In our work, we chose the widely used linear regressor, because it is easy to tune and well-behaved. Our linear regressors approximate Q-values as a linear combination of the features of the domain, with independent weights for each action. The approximation for state $s = (\phi_1, \ldots, \phi_n)$ and action a with parameters $(w_{a,0}, w_{a,1}, \ldots, w_{a,n})$ would be

$$\tilde{Q}(s,a) = w_{a,0} + \sum_{i=1}^{n} w_{a,i}\phi_i. \tag{10}$$

We did not use a classical batch version like the one described above, but an online implementation. Instead of running the regressor after having collected all available information from the domain, we make small updates to the linear approximation after each interaction with the environment. After taking action a in state $s = (\phi_1, \ldots, \phi_n)$ to obtain reward r and reach state $s' = (\phi'_1, \ldots, \phi'_n)$, all the weights for action a are updated with

$$w_{i,a} = w_{i,a} + \alpha \frac{\partial \tilde{Q}(s,a)}{\partial w_{i,a}} \left(r + \gamma max_{a' \in \mathcal{A}} \tilde{Q}(s', a') - \tilde{Q}(s,a) \right), \tag{11}$$

where

$$\frac{\partial \tilde{Q}(s,a)}{\partial w_{i,a}} = \begin{cases} 1 & \text{if } i = 0 \\ \phi_i & \text{if } i = 1, \ldots, n. \end{cases}$$

Online implementations are convenient and often the only option in large environments, since it may not be feasible to store all the transitions that need to be observed to obtain a good model. Additionally, by updating the model more frequently, the algorithm explores the interesting parts of the state space faster, and as the policy improves, the approximation focuses on accurately representing the part of the state space that is most often visited by the optimal policy.

3.2 LSPI

LSPI [43] is a stable algorithm that is more sample-efficient than fitted Q-learning, but it has a higher computational cost. LSPI is an actor-critic method that uses a linear approximation to the value function, like the one shown in (10).

We use three variants of LSPI: classic LSPI, IncLSPI, and LARS-TD. The original version of LSPI needs to store all the samples, which is impractical in large domains, and also needs a matrix inversion step of time complexity $O(n^3)$, where n is the number of features. This makes LSPI impractical for our domains when learning in the original state space, so in our experiments we use an incremental implementation with lower computational complexity that we call IncLSPI.

Our IncLSPI is similar to iLSTD [27], but in our implementation, we run a fixed behavioral policy and periodically update the policy [10] and reset the μ and A matrices to satisfy the assumptions made by iLSTD. Like iLSTD, IncLSPI is particularly efficient in sparse domains where the number of non-zero features k is small with respect to the total number of features n; however, it has complexity $O(n^2)$ in general.

LARS-TD [39] is an L1-regularized version of LSPI with complexity $O(mnk^3)$ where m is the number of samples and n and k are defined above. This algorithm selects features that are most likely to affect the function being approximated in order to make LSPI robust to stochastic features as well as more feasible in large domains. This set of features is actively modified by adding and removing features

until a chosen regularization criterion is satisfied. Our implementation of LARS-TD was modified to learn Q-functions.

3.3 Limitations of Linear Models

Linear architectures are usually easier to tune, analyze, and debug than others so they are a frequent choice for complex domains. However, a poor choice of features can make a linear representation useless. Imagine a simple grid domain with the coordinates of the agent and the coordinates of the goal position as features, and actions to move north, south, east and west. A linear architecture cannot represent a policy that leads the agent from an arbitrary cell to an arbitrary goal cell, because whether one action is preferred over another does not depend on any of the features of the state space, but on the relation between different features (the coordinates of the agent and the coordinates of the goal).

However, a linear architecture can use any number of nonlinear features based on any combination of the *native* features of the original state space. Therefore, any value function can be approximated if the right features are used. In the extreme case, we can imagine a single feature that outputs the true value function, and the linear approximation would be perfect by assigning a weight of one to that feature and zero to the rest. Unfortunately, finding the right features for complex domains can be a daunting task, often involving time-consuming trial and error. Furthermore, manual feature engineering is not an option for autonomous learning agents. For this reason, our attention focus algorithms try to subdivide the function needing approximation into smaller problems that will be more likely properly approximated by a linear architecture on the native feature space of the domain.

3.4 Other Algorithms

There are many other approximation algorithms [75] that we do not use in our work because their requirements or slow convergence limit their usefulness in general domains and make them unsuitable for our purposes. Nonlinear approximation architectures [63], for example, offer a larger hypothesis space, but they do not perform well in practice and require time-consuming manual parameter tuning. Some algorithms try to automate this process [80], but are not yet practical for large domains. Policy gradient approaches usually need an initial policy with decent performance, and are generally slower than value function approximation [7, 82, 74]. Residual gradient algorithms are also slow [46, 60].

Recent TD methods [73] offer attractive theoretical properties, but so far these methods are limited to prediction, *i.e.,* determining the value function V^π of a given policy π, as opposed to finding the optimal policy π^*. Greedy-GQ [49] solves this problem and can find an approximation of the optimal value function as long as the behavior policy is fixed. Unfortunately, in complex domains, a random behavior policy does not visit the interesting parts of the state space often enough for effective learning, and the algorithm is no longer stable with a varying behavior policy.

There are also efforts to adapt the KWIK[1] learning model to approximation architectures [47]. We believe this is a promising approach, but the current implementation assumes access to a perfect oracle that predicts Q-values with no error, which is not available in most domains.

[1]KWIK stands for *knows what it knows*; it is a family of PAC (probably approximately correct) algorithms for solving MDPs that includes R-max, discussed in Section 2.2.5.

CHAPTER IV

ABSTRACTIONS FOR REINFORCEMENT LEARNING

Abstraction is one of the most common ways of scaling up reinforcement learning, along with function approximation and often overlapping with it. There is a rich and varied literature on the topic, going from state-space abstractions that clump similar states together to hierarchical approaches that define either temporally-extended actions or task subdivisions. This chapter reviews previous RL abstraction approaches so we can later position our attention focus algorithms with respect to the existing literature.

4.1 Hierarchical Abstractions

Hierarchical RL attacks a learning problem by dividing it into different levels of complexity. To this end, the MDP is partially or completely divided into different subtasks that are easier to solve, and a higher-level process controls which subtask should be carried out at each moment. For example, an autonomous indoor vacuum cleaner could define subtasks such as *go to next room* and *clean current room*, and each subtask would be defined in terms of primitive actions: move forward, turn, suck dirt, etc.

MAXQ [17] is one of the best known hierarchical RL algorithms. In MAXQ, a system designer engineers a task hierarchy, and decomposes the reward to indicate which levels of the hierarchy are potentially responsible for the reward. MAXQ cannot find globally optimal policies, but *hierarchically optimal* ones, this is, the best policies that are consistent with the imposed hierarchy. However, it can speed up learning significantly by dramatically reducing the complexity of the policy that must be learned at each level of the hierarchy. HEXQ [31] extends MAXQ by using a heuristic,

based on the relative frequency of change of the features, to try to automatically build a MAXQ hierarchy. Unfortunately, this is only useful in a limited set of domains; for example, it would not help in a continuous domain where the values of all the features change at each step.

Other hierarchical algorithms try to reduce complexity by implementing temporally extended actions or macroactions [6] that represent sequences of actions. Then, a high-level policy can decide at each moment to perform, instead of a primitive action, one of these macro-actions, delegating control to the macro-action until a certain criteria is met. The most representative of this class of approaches is the *options* framework [71]. An option is a temporally extended action defined by a policy π that is followed while the policy is active, an initiation set $\mathcal{I}$ that defines the region of the state space where the action can be taken, and a termination condition β that provides the probability of the option exiting at any given state.

The original work on options assumes that the options are provided to the learning algorithm, including their policy. Several extensions try to learn options automatically using different heuristics such as relative novelty [86], bottleneck states [70, 52] or state clusters [50]. Other work focuses on learning options by providing the algorithm a per-option reward signal [34]. Other authors are critical of the options framework, arguing that options can harm learning due to pathological exploration patterns, even in simple domains, and that the speed-up associated with the options framework is a just a consequence of its implicit experience replay [33].

There are also promising algorithms based on intrinsically motivated learning [78], but so far these are restricted to domains where the agent can control all the variables of the state space.

Alternatives to hierarchies include a flat compositions of local models [76], but these algorithms need highly domain-specific knowledge that must also be specified by a system designer.

In general, hierarchical approaches to RL are useful and conceptually appealing, but their use in autonomous learning systems is difficult due to the necessity of either an engineer manually designing a task structure or the task fitting a specific heuristic that allows an algorithm to recover the task structure automatically.

4.2 State-Space Abstraction

State-space abstractions simplify learning by clumping together or generalizing among similar states. A state-space abstraction approach closely related with function approximation is *regularization*. Regularization is a technique widely used in machine learning that prevents overfitting by penalizing solution complexity. It is key for supervised learning in scenarios where there is little training data and a large hypothesis space, but its application to RL is not straightforward. In RL settings, regularization aims at determining which features of the state space are relevant, but this is challenging due to the noisy nature of the reward signal and the fact that, in RL problems, different features may be relevant at different times throughout the task. Nonetheless, previous work has combined RL with L_1 regularization [40], L_2 regularization [23], and "forward-selection" style feature selection from features generated based on Bellman error analysis [36, 59].

A different approach to state-space abstraction is to consider a set of predefined state-space representations and find the most useful one [41, 65]. Unfortunately, this approach is impractical in high-dimensional domains if there is no previous knowledge about useful representations or which features are useful together. In addition, these approaches pick a fixed representation for a given task, and they cannot shift the attention focus dynamically as our algorithm does. The U-tree [51] can find its own representations, but it requires too many samples to scale well in realistic domains.

4.3 Relational Reinforcement Learning

Yet another approach to abstraction is to use reinforcement learning on varying-length representations based on meaningful entities such as objects. This idea comes from the field of relational reinforcement learning [21, 58]. Typically, a human designer creates a detailed task-specific representation of the state space in the form of high-level facts and relations that describe everything relevant for the domain. These approaches offer great flexibility, but encoding all this domain knowledge for complex environments is impractical for autonomous agents that cannot rely on an engineer to provide them with a tailored representation for any new task they may face. Object-oriented MDPs (OO-MDPs) [18] constitute a related but more practical approach that sees the state space as a combination of objects of specific classes; unfortunately, OO-MDP solvers also need a designer to define a set of domain-specific *relations* that define how different objects interact with each other.

RMDP [29] solvers make assumptions similar to OO-MDPs, but require a full dynamic Bayesian network representation of the MDP. Fern proposed a relational policy iteration algorithm [24], but it relies on a resettable simulator to perform Monte-Carlo rollouts and also needs either an initial policy that performs well or a good cost heuristic for the domain.

CHAPTER V

ATTENTION FOCUS

The attention focus algorithms introduced in this thesis draw inspiration from humans to achieve significant speed-ups in reinforcement learning. As pointed out in Chapter 1, our brains receive, at any given moment, up to 11 million pieces of information, but we can be consciously aware of at most 40 of these [81]. In addition, adults can only hold a maximum of three to five meaningful items or "chunks" in their working memory [16]. This indicates that humans are able to cope with the complexity of the tasks they face using aggressive dimensionality reduction of the sensory information they receive. This dimensionality reduction enables them to quickly learn compact policies that are nearly optimal. This ability is called *selective attention*; it allows us, for example, to focus on a specific conversation in a crowded environment with many people talking at the same time.

Human selective attention is founded in elaborate and still not well understood neural circuits that have evolved over millions of years, as well as in a symbolic reasoning ability that enables humans to figure out what is important in order to accomplish a task. These foundations are not available today for our learning algorithms, so we have developed the idea of attention focus, in which we substitute these innate human capabilities with forms of domain information that are easy to acquire and generic enough to be useful for autonomous learning agents.

Our work explores two approaches to obtaining this kind of domain information, which we call attention focus. The first approach is to exploit the structure implicit in demonstrations of non-expert humans performing a task. In this case, we are copying the selective attention of humans when carrying out the task, inferring it from their

actions. The second approach is to apply generic world models in an attempt to see the world in a similar way as humans do. Instead of seeing the world as a set of meaningless features, these models consider the world as a set of independent objects, each one with its own features, that do not interact or interact weakly with each other.

Attention focus can be seen both as a function approximation (FA) and as a state abstraction. However, as we will see in Section 6.5.4, attention focus can be combined with previously existing FA and abstractions.

Previous work has already considered applying the concept of selective attention to learning. U-tree [51] and G [11] algorithms use an interesting approach to selective attention that first treats the state space as a single state. This state is later iteratively subdivided using a Kolmogorov-Smirnov test that decides which feature encodes the most important distinction. These methods do not need domain information, but it is impractical to find these subdivisions for large or continuous state spaces, especially without an initial good policy to aid exploration. RL has also been used for the problem of selective visual attention [54], but that is a different pursuit than using selective attention as a preprocessing step to enable RL in complex state spaces.

The rest of this chapter gives more detail about different sources of attention focus information, types of domains that can benefit from the approach, and the relation of attention focus with previous FA and abstraction algorithms.

5.1 Sources of attention focus information

We consider two different options to obtain attention focus information in order to speed up learning. The applicability of each approach depends on two factors, the type of information that is available and the specific characteristics of the domains. Chapter 8 studies how our two approaches can be combined, if the conditions for both approaches are met, for further learning speed-ups.

Regarding the type of information available, the first option is to obtain a set of

demonstrations of the desired task from human teachers and analyze the demonstrations to infer the features that are important to the teachers. The second option can be used when demonstrations are not available but the state space can be represented as a composition of different objects. Seeing the domain as a composition of different objects, we can make assumptions on how these objects interact with each other to infer which object should be the center of attention at each moment.

With respect to the specific characteristics of the domain, we can use attention focus by human demonstrations when there are some features of the domain that are irrelevant for a good policy and can be safely ignored, either in the entire domain or in subregions of the state space. The second option, using world models, applies to domains composed by objects that interact weakly with each other.

5.1.1 Human demonstrations

The most direct way of using attention focus in learning algorithms is to simply copy the selective attention of humans when they perform the task. One option would be to directly ask humans which features are important, but this is problematic for two reasons. First, even if humans had a clear and unbiased idea of what is important to them when performing a task, most people do not understand the internal representation of the state space in a learning agent, and therefore they will not be able to convey the information to the agent. Second, humans perform selective attention without being consciously aware of it [57], so they may not have truthful insight on which aspects are important to them when carrying out a task.

We sidestep those problems by obtaining all the necessary information from demonstrations by humans performing the task. With an adequate set of demonstrations, we can measure the mutual information between each feature of the state space (as the learning agent sees it) and the actions that the teachers take. With these measurements, we can determine which features are actually relevant to the teachers and

focus on those features in subsequent learning.

By reducing linearly the number of relevant features, we are reducing exponentially the number of states that need to be considered, and this dimensionality reduction will make subsequent learning algorithms more efficient. However, it may be the case that the desired task needs all the features of the state space, but the task can be decomposed into smaller subtasks, each one depending on a reduced number of features. Our algorithms for attention focus from human demonstrations are also able to deal with this situation. They use the demonstrations to figure out, at the same time, a task decomposition and an adequate state-space abstraction for each subtask.

In this category, we have developed two algorithms, abstraction from demonstration (AfD) [15] and automatic decomposition and abstraction from demonstration (ADA) [13]. Chapter 6 gives a complete description of these algorithms, along with experimental evaluation of their performance.

5.1.2 World models

We have also developed attention focus algorithms adequate for tasks for which it is not practical to obtain demonstrations, but for which an object-oriented representation of the domain is available. This representation can be native or can be generated from native features with automatic techniques, *e.g.,* from a pixel image using off-the-shelf vision software such as OpenCV [8].

For these approaches, we assume that the objects interact weakly among themselves and can therefore be modeled separately, with the total value function being a composition of the value function with respect to each object. Modeling each object separately leads to an exponential reduction in the number of states, compared to modeling the state space as a whole, and it allows the agent to learn object-specific policies quickly. Then we use a model of attention to determine which objects should be the in focus of attention, *i.e.,* determine the action to be taken, at each moment.

Note that while we are using a world model of separate objects that interact weakly among themselves, we are not trying to recover the underlying MDP model, nor object-specific MDP-models, but just object-specific policies. For this reason, our algorithms are still model-free reinforcement learning algorithms.

Chapter 7 describes and empirically evaluates OF-Q [14], our "attention focus from world models" algorithm. We first develop a version that assumes complete independence between the different objects, and later extend it to the more general case where objects can influence each other.

5.2 Application Domains

Attention focus algorithms exploit common patterns in typical real-world tasks, trying to imitate the tricks that humans use to leverage these patterns. Obviously, there is no perfect solution for the curse of dimensionality, which means that attention focus algorithms will not be useful in arbitrary domains. For example, a domain where the optimal action depends on an arbitrary function that depends on all the features of the domain would not benefit from our methods.

However, attention focus methods can extend reinforcement learning to handle many real-world tasks that could not be tackled before because of their high dimensionality. Specifically, we focus on tasks that humans can carry out easily, but can be challenging for a computer. As we have seen, humans quickly find policies for complex task by performing aggressive feature selection on their sensory input. With such a strong simplification of the environment, humans do not look for optimal policies but for *satisficing* ones [66]. Satisficing is a portmanteau of the words satisfy and suffice, *i.e.,* satisficing policies are policies that are close enough to an optimal policy for the incentive of the optimal policy to offset the cost of finding it.

Attention focus algorithms also aim for satisficing policies for two reasons. First,

we will often face NP-hard problems, like many videogames [3] or a simple transformation to an MDP of a travelling salesman problem. This means that, unless $P = NP$, there is no option but to sacrifice optimality for the sake of efficiency in complex domains. The second reason is that, often, a satisficing policy is more convenient than the optimal one because it can transfer better to slightly different versions of the same task. A perfect optimal policy for solving a level of a game of Pacman is not going to be useful for a different level. However, a policy that uses patterns in the area of the screen close to the acting agent may not be optimal but, if enough neighborhoods are recognized, it will likely work with any level. In short, satisficing policies are necessary for complex domains, and they are also beneficial for transfer learning.

With respect to representation, attention focus algorithms are most useful with state representations relative to the agent and deictic representations [25]. In general, it is fair to assume that these representations are native representations because this will be the case for embodied agents that perceive the world through a series of sensors. It is also the natural representation for humans.

Successful applications of attention focus need either demonstrations or an object-based view of the world. Given that we target tasks that humans can carry out and the availability of computer vision libraries, this is a fair assumption for the tasks we are interested in.

5.3 Relation to Other Function Approximation Algorithms

Attention focus algorithms can be considered as function approximation approaches because they approximate the true value function by either collapsing states together (so they share a Q-value that is not necessarily the same) or assuming that the value function is a composition of the value functions of the different objects in the domain. This is different, however, from traditional FA, which imposes a specific functional

form (*e.g.,* a linear model on the features) on the value or Q-function.

Another difference between attention focus and other FA approaches is that most FA techniques assume that a suitable set of features is provided, so that the combination of the features with the model provides a hypotheses close enough to the true value function. Attention focus approaches, however, start from the original feature space of the problem and try to improve it by eliminating unneeded features and subdividing either the state space or the value function into simpler components. By eliminating unneeded features, it reduces the complexity of any algorithm that would have had to consider these features. By dividing the problem into simpler components, attention focus makes it more likely that any feature representation and model will be able to represent each of the components.

For these reasons, as we show in Section 6.4, attention focus can be combined with previously existing state-of-the art FA approaches to increase their effectiveness. Therefore, attention focus can be used to push the limits on the complexity of the domains for which we can learn a satisficing policy with reinforcement learning. This is an important result because FA algorithms perform implicit feature selection that might have overlap with the feature selection that attention focus provides, and these would render irrelevant the benefits that attention focus provides.

5.4 Relation to Other Abstraction Algorithms

Attention focus from demonstrations can also be seen as a state abstraction algorithm because it clumps together similar states into a single, abstract state. When the demonstrations are also used to split a problem into different subtasks, it also takes the role of a hierarchical abstraction.

The most similar previously-existing state abstraction is regularization, introduced in Section 4.2. Like attention focus from demonstration, regularization tries to limit the complexity of the solution (in this case, a Q-function) by figuring out irrelevant

features. However, using human selective attention allows a more aggressive state abstraction than regularization, because the policy to be approximated is not the optimal one but a satisficing human-like policy based on a small set of features.

Another interesting point is that the use of attention focus is compatible with temporally extended actions frameworks such as options [71]. In fact, by reducing the complexity of the state space, attention focus can facilitate the task of learning automatically the policies for such macro-actions.

CHAPTER VI

ATTENTION FOCUS FROM HUMAN DEMONSTRATIONS

This chapter describes our algorithms for attention focus from human demonstrations, and experimentally measures the performance that they offer. The first algorithm, abstraction from demonstration (AfD) [15], uses a set of human demonstrations H to build a state-space abstraction that will speed up subsequent learning. The second algorithm, automatic decomposition and abstraction from demonstration (ADA) [13], extends AfD to subdivide a task in smaller subtasks where we can perform a more aggressive and fine-grained state abstraction. In the hierarchy of MDP state abstractions [48], AfD/ADA abstractions are in between a^*-irrelevance and π^*-irrelevance.

6.1 Additional Notation

Human demonstrations are defined as a set of episodes, each comprising a list of state action pairs

$$H = \{\{(s_1, a_1), (s_2, a_2), \ldots\}, \ldots\}, s_i \in S, a_i \in A.$$

We define

$$\vec{mi}_E = (I(F_1; A), \ldots, I(F_n; A))$$

as a vector whose elements are the mutual information (see Appendix A) between each feature of the state space and the action taken by the human teacher, according to the samples of H in the region $E \subset S$. To compute $\vec{mi}_E$, we estimate the appropriate joint and marginal probability density functions using the samples of H that fall in region E, and use the results with (18).

6.2 Abstraction from Demonstration

6.2.1 Algorithm

Abstraction from demonstration (AfD) learns a policy for an MDP by building an abstract space S^α and using reinforcement learning to find an optimal policy that can be represented in S^α. AfD obtains S^α by selecting a subset of features from the original state space S with which it can predict the action that a human teacher has taken in a set of demonstrations H. Learning in S^α can be significantly more efficient because a linear reduction in the features leads to an exponential reduction in the size of the state space.

AfD, shown in Algorithm 1, is composed of two sequential steps. First, a feature selection algorithm is applied to human demonstrations to choose the subset of features we will use. In the second step, which corresponds to the loop in Algorithm 1, the algorithm learns a policy for M using a modified version of a Monte-Carlo learning algorithm with exploring starts. Instead of learning the Q-values of states in the original state space $Q(s, a), s \in S$, it learns the Q-values of states in the transformed state space $Q(s^\alpha, a)$, where $s^\alpha \in S^\alpha$. We stop when policy performance converges, *i.e.,* when the expected discounted reward remains stable.

We have used two different feature selection algorithms for AfD. The first, which we call C4.5-greedy, is a simple greedy backward selection algorithm. Starting with the full feature set, it iteratively removes features, one at a time, that have little impact on performance. In each iteration, the feature whose absence affects accuracy the least is removed. If the best feature to drop affects accuracy by more than 2% with respect to the current feature set, we stop. We also stop if dropping a feature results in an accuracy loss greater than 10% with respect to the original feature set. These stopping parameters do not appear to be sensitive. In experiments we have tested parameter values of 1% and 5% and found no significant difference in the feature set selected. Note that we use *relative accuracy* for these stopping criterion, *i.e.,* the

Algorithm 1 Generic AfD algorithm with $\gamma = 1$; the general case is a simple extension.

Require: MDP $M = (S, A, P^a_{ss^\alpha}, R^a_s, \gamma)$, $S = \{F_1 \times \ldots \times F_n\}$, feature selector F, human demonstrations $H = \{\{(s_1, a_1), (s_2, a_2), \ldots\}, \ldots\}, s \in S, a \in A$.

```
chosenFeatures ← F(H)
π ← arbitrary policy
Initialize all Q(s^α, a) to arbitrary values
Intialize all Rewards[(s^α, a)] to []
while π performance has not converged do
    visited ← []
    Start episode with random state-action, then follow π
    for all Step (state s, action a, reward r) do
        for all (s^α, a) in visited do
            EpisodeReward[(s^α, a)] += r
        end for
        s^α ← getFeatureSubset(s, chosenFeatures)
        if (s^α, a) not in visited then
            Add (s^α, a) to visited
            EpisodeReward[(s^α, a)] ← 0
        end if
    end for
    for all (s^α, a) in visited do
        Add EpisodeReward[(s^α, a)] to Rewards[(s^α, a)]
        Q(s^α, a)=average(Rewards[(s^α, a)])
    end for
    Update greedy policy π
end while
```

amount of accuracy gained with respect to the majority classifier.

The second feature selection algorithm, Cfs+voting, uses Correlation-based Feature Subset Selection (Cfs) [30]. Cfs searches for features with high individual predictive ability but low mutual redundancy. The Cfs algorithm is used separately on the demonstrations of each teacher. A feature is chosen if it is included by at least half of the teachers.

6.2.2 Policy invariance

State abstractions that, like AfD, collapse states with the same optimal action are called policy-invariant. These abstractions can be problematic; even if they can represent a policy, they might not be good enough to learn it when using bootstraping algorithms such as Q-learning or Sarsa.

In the transformed state space, a single state $s^\alpha \in S^\alpha$ represents several states of the original space $s_1, \ldots, s_k \in S$. Assuming the subset of features selected is sufficient for predicting the teacher's action, every original state corresponding to s^α should share the same action. Because only states with the same associated action are collapsed, the teacher policy can be represented in S^α. The original state value function $V(s)$ may not be representable in S^α because the collapsed states $s_1, \ldots, s_k$ may have different true state values in S, but in S^α they all share the state value of s^α. The same holds for the Q-values $Q(s, a)$.

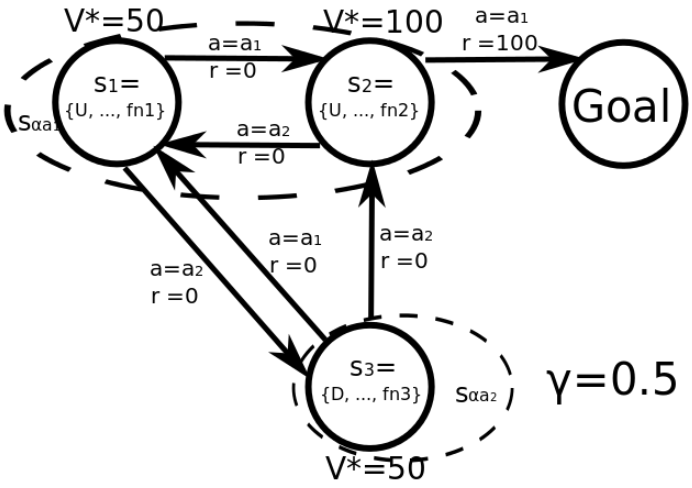


Figure 5: Example of an MDP that loses the Markov property upon state abstraction. Circles represent states in S and arrows transitions when taking action a, providing an immediate reward r. Dashed circles represent states in the transformed state space S^α.

For example, Figure 5 shows a deterministic MDP with states defined by features $\{F_1, \cdots, F_n\}$. F_1 can have values U or D depending whether the state is up or down in the figure, and the rest of the features are policy-irrelevant. The available actions are a_1 and a_2, and the only reward is obtained by taking action a_1 in state s_2. With $\gamma < 1$, the value of s_1 and s_2 is different; however, because their optimal action is the same (a_1), AfD collapses both into a single state $s^\alpha_{a_1} \in S^\alpha$.

Conversion to the transformed space also breaks the Markov property. In Figure 5, we can see that in the transformed space, actions a_1 and action a_2 in collapsed state s^α_{a2} are identical. However, we know that the two actions are very different. The value of taking action a_1 from collapsed state $s^\alpha_{a_1}$ depends upon how we got to $s^\alpha_{a_1}$; specifically, what action was taken in state $s^\alpha_{a_2}$ to get there. This means reinforcement learning algorithms that use bootstrapping will not work. Bootstrapping algorithms such as Q-learning compute Q-values considering the immediate reward and the value

of the next state. In our example, with the immediate reward and next state being identical for both actions, Q-learning would be unable to learn the best action, even though one leads to higher discounted rewards.

Previous work on policy-invariant abstractions [32] encapsulates the state abstraction as an option that will not be used if it degrades performance. We work around problems of policy-invariant abstractions by using non-bootstrapping methods like Monte Carlo.

6.2.3 Theoretical properties of AfD

AfD's feature elimination process is benign: in the limit of infinite data, the feature subset it yields will not negatively affect the accuracy of the learner. To see this, consider that AfD only removes features that it judges will not reduce the accuracy of the learner. These judgements are based on some held out portion of the data.[1] In the limit of infinite data, these judgements will be accurate, and the elimination process will never remove a feature if it negatively impacts accuracy. Thus, the final feature subset cannot negatively impact accuracy.

Similarly, AfD's policy solver is sound. As shown in Section 6.2.2, in a policy-invariant abstraction like ours, bootstrapping algorithms like Q-learning may not find the best stationary policy; however, the abstracted space creates a POMDP, where Monte-Carlo control is sound [67]. Thus, applying the policy solver will not lower policy performance.

From these properties, we can show that, in the limit, the worst-case policy performance of AfD is the same as direct LfD. In the limit, LfD will yield a policy with the best prediction accuracy.

[1] We actually use cross-validation for higher sample efficiency, but the same principle holds.

6.3 Automatic Decomposition and Abstraction from Demonstration

It is often the case that multipurpose agents have a high number of input signals of which only a subset are relevant for any specific task; however, using AfD does not help if all state features are relevant for the task to be learned.

Our key insight is that there are often tasks where all features are relevant for some part of the task, but the task can be decomposed into subtasks, such that for any given subtask, there exists an abstraction in which the policy can be expressed. For example, when we drive a car, we focus our attention almost completely on the car keys at the start and end of a drive, but completely ignore them for the rest of the trip.

Our goal is to infer this shifting selective attention, the particular task decomposition and state abstraction that the human teacher uses during her demonstration. We define a subtask as a region of the state space where only a subset of features is relevant, with this subset differing from those of other subtasks. Thus, we look for a decomposition that maximizes our ability to apply AfD in each part.

Our algorithm, *automatic task decomposition and state abstraction from demonstration* (shortened to automatic decomposition and abstraction, ADA), uses human demonstrations to both decompose a task into its subtasks and find independent state abstractions for each subtask. ADA can build more powerful abstractions than AfD, finding compact state-space representations for more complex tasks in which all state features are relevant at some point in the task.

To determine which features are relevant to a particular subtask, we measure the mutual information between each state feature and the action taken by a human teacher in a set of demonstrations. Once the state space is decomposed into different subtasks, the agent can learn and represent a compact policy by focusing only on the features that are relevant at each moment.

6.3.1 Algorithm overview

Given an MDP M and a set of human demonstrations H for a task to be learned, ADA finds a policy in three conceptual steps and an optional fourth step:

1. **Problem decomposition.** Using H, partition the state space S into different subtasks $T = \{t_1, t_2, \ldots\}$, $\cup T = S$, $t_i \cap t_j = \emptyset$ if $i \neq j$.

2. **Subtask state abstraction.** Using H, determine, for each subtask $t_i \in T$, the relevant features $\hat{F}_i = \{F_{i_1}, F_{i_2}, \ldots\}, F_{i_j} \in F$, and build a projection from the original state space S to the abstract state space $\phi(s) = \{i, f_{i_1,s}, f_{i_2,s}, \ldots\} \in S^\alpha, s \in t_i$.

3. **Policy construction.** Build a stochastic policy $\pi(s^\alpha)$, projecting the samples from H into S^α.

4. **Policy improvement (optional).** Use reinforcement learning to improve the policy found in the previous step. We refer to our algorithm as ADA+RL when it includes this step.

We list the first two steps as if they were sequential; however, they are interwoven and concurrent. The decomposition of the state space depends on the quality of the abstractions that can be found on different subspaces. We describe them as separate steps just to make the algorithm description more straightforward.

6.3.2 Problem decomposition

Definition 1. *A set of* ***subtasks*** $T = \{t_1, t_2, \ldots\}$ *of an MDP M is a set of regions of the state space S such that:*

- *The set of all subtasks T forms a partition of the original state space S, i.e., $\cup T = S$ and $t_i \cap t_j = \emptyset$ if $i \neq j$.*

- *A subtask t_i is identified by having a local satisficing policy π_i that depends only on a subset of the available features. This subset is different from neighboring subtasks.*

- *The global policy $\pi(s^\alpha)$, combination of the policies of each subtask, is also satisficing.*

While this definition is not the typical one for subtasks in a sequential decision problem, it turns out to be a useful one, particularly if we focus on human-like activities. For example, cooking an elaborate recipe requires multiple steps, and each of these steps will involve different ingredients and utensils. It is possible that two conceptually different subtasks may depend on the same features, but in our framework, and arguably in general, the computational benefits of separating them are not significant.

With ADA, we can identify these subtasks given a set of human demonstrations H with two requirements:

- There must be a sufficient number of samples min_{ss} from each subtask in the set of demonstrations H.

- The class of possible boundaries between subtasks $B = \{b_1, b_2, , \ldots\}, b_i \subset S$ must be defined. Each boundary divides the state space in two, b_i and $S - b_i$. ADA will be able to find subtasks that can be expressed as combinations of these boundaries.

The necessary number of samples is determined in the first step of the ADA algorithm. This minimum sample size is needed due to the metric we use to infer feature relevance. Mutual information is sensitive to the *limited sampling bias*, and will be overestimated if the number of samples considered is too low.

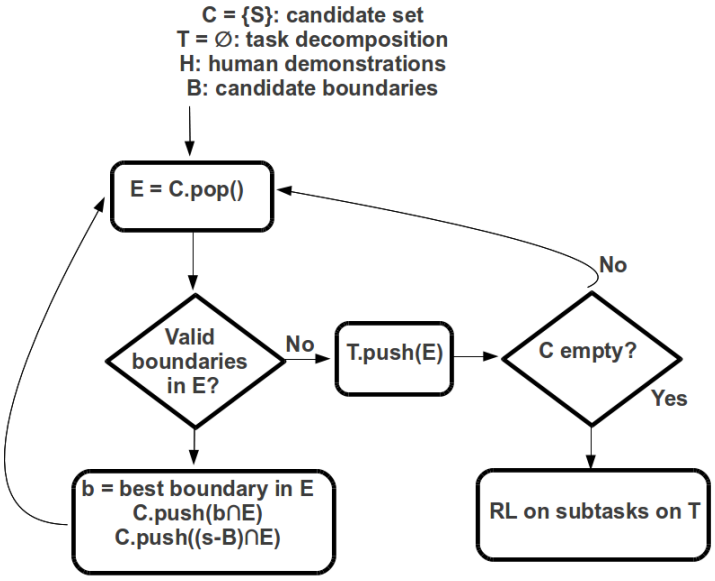


Figure 6: Diagram of automatic decomposition and abstraction from demonstration.

Algorithm 2 ADA problem decomposition.

Require: MDP $M = \left(S, A, P^a_{ss'}, R^a_s, \gamma\right)$, $S = \{F_1 \times \ldots \times F_n\}$, human demonstrations $H = \{\{(s_1, a_1), (s_2, a_2), \ldots\}, \ldots\}, s \in S, a \in A$, boundaries $B = \{b_1, b_2, \ldots\}, b_i \in S$, ϵ.

```
min_ss ← min_sample_size(H, ε)
T ← {}
𝕊 ← {S}
while 𝕊 ≠ ∅ do
    {pop removes the element from 𝕊}
    E ← 𝕊.pop()
    B_E ← {b ∈ B, valid_split(b, E, min_ss)}
    if B_E = ∅ then
        T.push(E)
    else
        b_best ← arg max_{b∈B_E}(boundary_score(b, E)))
        𝕊.push(b_best ∩ E)
        𝕊.push((S − b_best) ∩ E)
    end if
end while
Return T
```

The decomposition is described in Algorithm 2 and sumarized in Figure 6. At each iteration of the while loop, we consider a subspace E, with $E = S$ in the first iteration. We then consider all valid boundaries. If there are none, then E itself is a subtask. If there are valid boundaries, we score them and choose the one with the highest score.

We then split E according to the boundary and add the two new subspaces to the list of state subspaces to be evaluated, so they will be further decomposed if necessary.

The boundaries B can have any form that is useful for the domain. In our experiments, we consider thresholds on features, *i.e.,* axis-aligned surfaces. Using these boundaries, Algorithm 2 is just building a decision tree with a special split scoring function and stopping criteria.

The following subsections discuss the details of the split scoring function, the discriminator of valid boundaries, and the estimator of the minimum number of samples. These contain the most interesting insights about ADA.

6.3.2.1 Boundary discriminator

Given a subspace $E \subset S$, m_{ss} and H, we consider a boundary $b \subset S$ to be valid if it meets three conditions:

1. There are enough samples in the set of human demonstrations H to ensure we can measure mutual information with accuracy on both sides of the boundary, *i.e.,*

$$|\{\{s,a\} \in H, s \in b \cap E\}| > min_{ss},$$
$$|\{\{s,a\} \in H, s \in (S-b) \cap E\}| > min_{ss}.$$

2. At least on one side of the boundary, either $b \cap E$ or $(S-b) \cap E$, it is possible to find a state abstraction, *i.e.,* some features are policy-invariant and can be ignored. We detail how we find these features in Section 6.3.3.

3. The state abstractions at each side of the boundary are not the same.

This boundary discriminator works as the stopping criteria of the algorithm. When there are no more valid boundaries to be found, the decomposition step finishes.

6.3.2.2 Boundary scoring

The boundary scoring function determines the quality of b as a boundary between different subtasks within a region $E \in S$:

$$boundary\_score(b, E) = \left|\left| \frac{\vec{mi}_{b \cap E}}{\|\vec{mi}_{b \cap E}\|} - \frac{\vec{mi}_{(S-b) \cap E}}{\left|\left|\vec{mi}_{(S-b) \cap E}\right|\right|} \right|\right|. \tag{12}$$

The score is the Euclidean distance between the normalized mutual information vectors on both sides of the boundary.

We are therefore measuring the difference between the relative importance of each feature on both sides of the boundary. Since we want to find subtasks that rely on different features, we choose the boundary that maximizes this difference (see Algorithm 2).

6.3.2.3 Minimum samples

Due to the *limited sampling bias*, mutual information is overestimated if it is measured with an insufficient number of samples. The minimum number of samples in ADA, given a set of demonstrations H and parameter $\epsilon \approx 0.1$, is

$$m_{ss} = \arg\min_{n} average \left(\frac{\vec{mi}_S - \vec{mi}_{S,n}}{\vec{mi}_S} \right) < \epsilon, \tag{13}$$

where $mi_{S,n}$ is the mutual information vector on the original state space S, taking only a subset of n randomly chosen samples from all the samples in H. The subtraction and division are elementwise and the average function takes the average of the values of the resulting vector. Because of the variability of mutual information, it is necessary to evaluate (13) several times for each possible n, each time with a different and independently chosen set of samples. Because of the limited sampling bias, the difference between $\vec{mi}_S$ and $\vec{mi}_{S,n}$ will grow as n decreases, and a binary search can be used to find m_{ss} efficiently.

6.3.3 Subtask state space abstraction

Given a region of the state space $E \subset S$, we consider $\vec{mi}_E$ in order to estimate policy-irrelevant features. Even if a feature is completely irrelevant for the policy in a region of the state space, its mutual information with the action will not be zero due to the limited sampling bias. Therefore, ADA groups the values of $\vec{mi}_E$ into two clusters separated by the largest gap among the sorted values of the vector. If the value difference between any two features in different clusters is larger than the distance within a cluster, we found a good abstraction that discards the features in the lower value cluster.

Note that this step occurs concurrently with the previous one, since the decomposition step needs to know in which regions of the state space there are good abstractions. Once these steps complete, we can build the projection function from the original state space S to the abstract state space $\phi(s) = \{i, f_{i_1}, f_{i_2}, \ldots\} \in S^\alpha, s \in t_i$.

6.3.4 Policy construction

Once the task decomposition and state abstraction are completed, and we have the projection function $\phi(s)$, we use the demonstrations H to build a stochastic policy that satisfies

$$P(\pi(s^\alpha) = a_i) = \frac{\left|\{\{s, a_i\} \in H, \phi(s) = \hat{s^\alpha}\}\right|}{\left|\{\{s, a\} \in H, \phi(s) = \hat{s^\alpha}, a \in A\}\right|}, \tag{14}$$

where $\hat{s^\alpha}$ equals s^α if $|\{\{s, a\} \in H, \phi(s) = s^\alpha, a \in A\}\}| > 0$. Otherwise, $\hat{s^\alpha}$ equals the nearest neighbor of s^α for which the denominator in (14) is not zero.

To compute the policy, we project the state of each sample of H into the abstracted space and make a normalized histogram of each action. This concludes the basic ADA algorithm.

6.3.5 Policy improvement

ADA+RL adds another step, policy improvement, in which we use reinforcement learning techniques to find the optimal policy that can be represented in the abstract state space S^{α}. Unlike traditional LfD techniques, ADA was designed so that the resulting policy can be easily improved given additional experience. In this way, we can potentially obtain a better policy than that of the human teacher.

Given the kind of abstraction that ADA performs, bootstrapping methods such as Sarsa or Q-learning are not guaranteed to converge in the abstract state space [48]. As such, we can use either Monte-Carlo methods or direct policy search.

6.4 Combination with Function Approximation

AfD and ADA are able to derive the correct attention focus using a small set of demonstrations, yielding exponential speed-ups in learning. However, the performance of both techniques was first measured only with tabular state-space representations. FA algorithms perform implicit feature selection that may overlap with attention focus, so it was initially unclear whether the benefits of attention focus would still be significant when combined with FA.

Further experiments have shown that ADA and AfD combined with function approximation offer significant advantages over FA alone. In particular:

- Attention focus derived from human demonstrations can be used with function approximation architectures to obtain near-optimal policies.
- Attention focus enhances the performance and applicability of RL algorithms with function approximation in the following ways:
  - Scalability, *i.e.,* making these algorithms able to deal with domains that were previously out of their scope due to high-dimensional state spaces.

– Speeding up learning. The speed-up grows with the scale of the problem and is large enough to justify the cost of acquiring demonstrations.

– Extending the hypothesis space of function approximation architectures, broadening the class of problems that such architectures can solve.

– Making the algorithms more automatic. Attention focus can make manual feature engineering less necessary by finding a good state-space representation. This allows agents to learn new tasks without intervention from an engineer.

We justify these claims experimentally in Section 6.5, measuring the effect of attention focus on two popular function approximation algorithms, fitted Q-learning and LSPI on three different videogame domains.

6.5 Experimental Evaluation

This section details the experiments we carried out to measure the performance of our algorithms. We first introduce our experimental domains and then analyze the performance, using a tabular representation, of AfD and ADA. Finally, we analyze the speed-up obtained when combining our attention focus algorithms with previous function approximation methods.

6.5.1 Domains

We implemented several videogame domains to test our algorithms. Some of these domains have a clear subtask structure, while others do not, so we can highlight the differences between AfD and ADA.

6.5.1.1 Pong

We used this simple domain for an early test of the soundness of AfD. Pong is a form of tennis where two paddles move to keep a ball in play. Our agent uses one

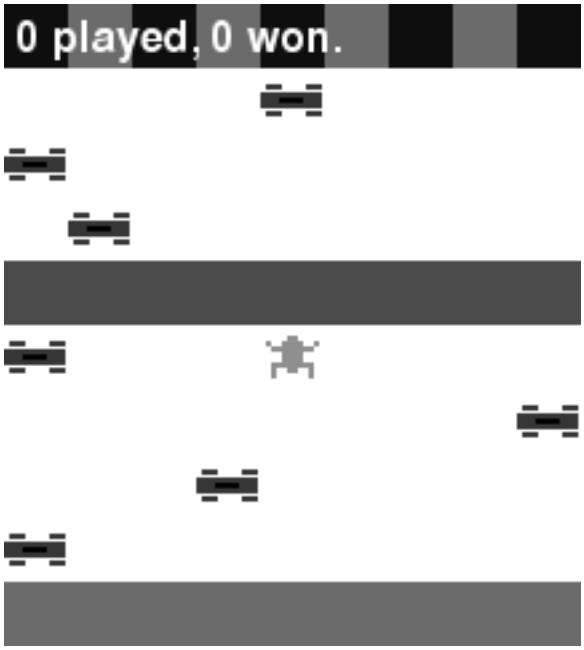


Figure 7: Screen capture of the Frogger domain.

paddle while the other paddle follows a fixed policy to move in the direction that best matches the ball's Y position when the ball is approaching, moving randomly otherwise. There are five features: `paddle-Y`, `ball-X`, `ball-Y`, `ball-angle`, and `opponent-Y`. `Y` coordinates and `ball-angle` have 24 possible values while `ball-X` has 18. There are two possible actions: `Up` or `Down`. The reward is 0, except when successfully returning a ball, yielding +10. The game terminates when a player loses or after 400 steps, implying a maximum policy return of 60.

6.5.1.2 Frogger

Our non-hierarchical high-dimensional domain is a version of the classic Frogger game (Figure 7). In the game, the player must lead the frog from the lower part of the screen to the top, without being run over by a car or falling in the water.

At each time step, cars advance one position in their direction of movement, and the player can leave the frog in its current position or move it up, down, left or right. The positions and directions of the cars are randomly chosen for each game, and the

frog can be initially placed in any position in the lower row. The game was played at 7 steps per second, chosen empirically as a challenging enough speed for the game to be fun.

The screen is divided into a grid, and the state features are the contents of each cell relative to the current position of the frog. For example, the feature `3u2l` is the cell three rows up and two columns to the left of the current frog position, and the feature `X1r` the cell just to the right of the frog. The possible values are `empty`, if the cell falls out of the screen; `good`, if the cell is safe; and `water`, `carR`, and `carL` for cells containing water, or a car moving to the right or left. There is a positive reward $r = 1000$ for reaching a goal cell and a negative reward $r = -100$ for failure. The discount factor is $\gamma = 0.99$.

There are 8x9 cells, so 306 features are needed to include the screen in the state representation. With 5 possible actions and 5 possible values per cell, a table-based Q-learning algorithm might need to store up to $5^{307} \approx 10^{215}$ Q-values.

As a comparison, the estimated number of atoms in the observable universe is just 10^{80}. This means the problem is intractable in the raw state space, and approaches that use human demonstrations as policy priors [38] may not be directly applicable.

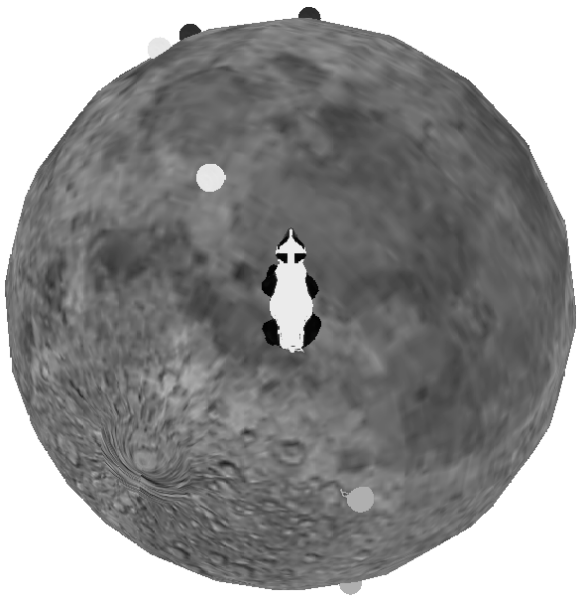


Figure 8: RainbowPanda domain. The agent must pick all balls, aiming at those that match the agent color at each moment. With 12 continuous state features, traditional RL does not converge in a reasonable amount of time. ADA finds a satisficing policy quickly by decomposing the problem into simpler subtasks.

6.5.1.3 Panda domains

To measure the effect on FA of the subtask decomposition of ADA, we used two different domains in which the player controls an agent represented as a panda bear walking on a spherical surface. The agent has to collect balls of different colors, as shown in Figure 8. At each time step, the panda can turn right/left or move forward/backward. The agent moves slower backward than forward. The position of the balls is chosen randomly at the start of each episode.

In the PandaSequential domain, there are several balls of different colors to be picked up in a specific fixed order. In the RainbowPanda domain, there are six balls, two of each possible color, and at any given time the panda is tinted by the color of the balls it must pick up. The active color always changes when there are no balls of the active color left and it may also change, with a small probability, at any time

step. The state features are the angle and distance of each ball relative to the agent and, for RainbowPanda, the active color. The agent receives a reward $r = 1000$ for each ball collected and there is a discount factor $\gamma = 0.99$.

In RainbowPanda, there are 12 continuous variables (relative angle and distance of each ball) and one discrete variable, the color the agent is currently allowed to pick up. In this 13-dimensional state space, traditional tabular RL takes an unreasonable amount of time to converge. Further, the complexity of the policy grows exponentially with the number of balls. We will show that with ADA we can automatically decompose this problem into a set of subtasks, one per color, with each one needing to pay attention only to the closest ball of the target color. These two-dimensional policies are easy to obtain, and the complexity of the global policy grows linearly with the number of balls.

6.5.2 Results on non-hierarchical domains

6.5.2.1 Setup

We compare the performance of AfD with that of using demonstrations alone and that of using RL alone (using Sarsa(λ)). For direct LfD, we use a C4.5 [61] decision tree classifier to learn a direct mapping. The classifier uses all the features of the state space as input attributes and the actions of the particular domain as labels.

Pong serves as a proving ground for demonstrating the correctness of AfD, using just 23 episodes as training data. The more complex Frogger gauges generalization and real-world applicability. For Frogger, we recruited non-expert human subjects—six males and eight females—to provide demonstrations. Each had three minutes to familiarize themselves with the controls of the game; they were then asked to provide demonstrations by playing the game for ten minutes.

Table 1: Performance on Pong. Time is measured in seconds.

Player	**Average Return**	**Episodes**
Human	56.5	–
Sarsa	60.0	2554
Direct LfD	15.7	–
AfD - C4.5-greedy	60.0	59

6.5.2.2 Results

Table 1 compares the performance of various learned policies in Pong. Human results are provided for reference. As we can see from the results, AfD is able to learn an optimal policy, outperforming direct LfD; moreover, while RL also learns an optimal policy, AfD is significantly faster. AfD's speed-up corresponds directly to the smaller abstract state space. In particular, AfD ignored the feature `opponent-Y`, which did not influence the teacher's actions.

With Frogger, we focus on how well AfD works with non-expert humans. The 14 human teachers obtained a success rate (percentage of times they lead the frog to the goal) between 31% and 55%. For learning in AfD, we filtered the demonstrations to keep only successful games and to remove redundant samples caused by the player not pressing keys while thinking or taking a small break. Each user provided on average 33.1 demonstrations ($\sigma = 9.3$), or 1230.1 samples ($\sigma = 273.6$). Note that a demonstration is a complete episode of the game and a sample is a single state-action pair.

We compared the algorithms using two sets of the demonstrations: the aggregated samples of all users, 464 demonstrations (17221 samples) in total, and the 24 demonstrations (1252 samples) from the best player.

For feature selection in AfD, we used the Cfs+voting and C4.5-greedy algorithms (Section 6.2.1). Before using C4.5-greedy in this domain, we reduced the number of variables using the Cfs algorithm, because iterative removal on 306 variables was too time consuming and our tests showed that the chosen features were the same as

Table 2: Frogger domain, all demonstrations (17221 samples).

Player	Success rate
Human	31%-55%
Direct LfD	43.9%
AfD - Cfs+voting	97.0%
AfD - C4.5-greedy	97.4%

Table 3: Frogger domain, best player demonstrations (1252 samples).

Player	Success rate
Human	55%
Direct LfD	17.1%
AfD - C4.5-greedy	88.3%

when using only C4.5-greedy. Cfs+voting was not used on the second data set, as it is designed to work with a set of demonstrations from different teachers.

Table 2 shows that, using all demonstrations, AfD achieved a significantly higher success rate than direct LfD. AfD enjoyed close to 100% success regardless of the feature selection algorithm used, while direct LfD did not reach 44%. Note also that the AfD policies performed better than the best human.

Table 3 shows results using only demonstrations from the best player. Even with only 7% the number of demonstrations of the previous experiment, AfD performance decreased only slightly. By contrast, direct LfD is much worse. Again, AfD performed better than the teacher. Comparing both tables, we can appreciate that AfD was much more sample efficient than LfD, performing better with 20 times fewer demonstrations.

By inspection, we see that AfD identified "key features" of the domain (the ones that would be included in a hand-crafted set) using either of the two datasets. The five key features for this domain are the cells at both sides of the frog and the three closest cells in the row immediately above. Of the original 306 features, the algorithms selected 9 to 12, and the five key features were included in these sets. Only when using just the best player demonstrations AfD did fail to include one of these key features, to which we attribute the slight decrease in performance. The other features selected

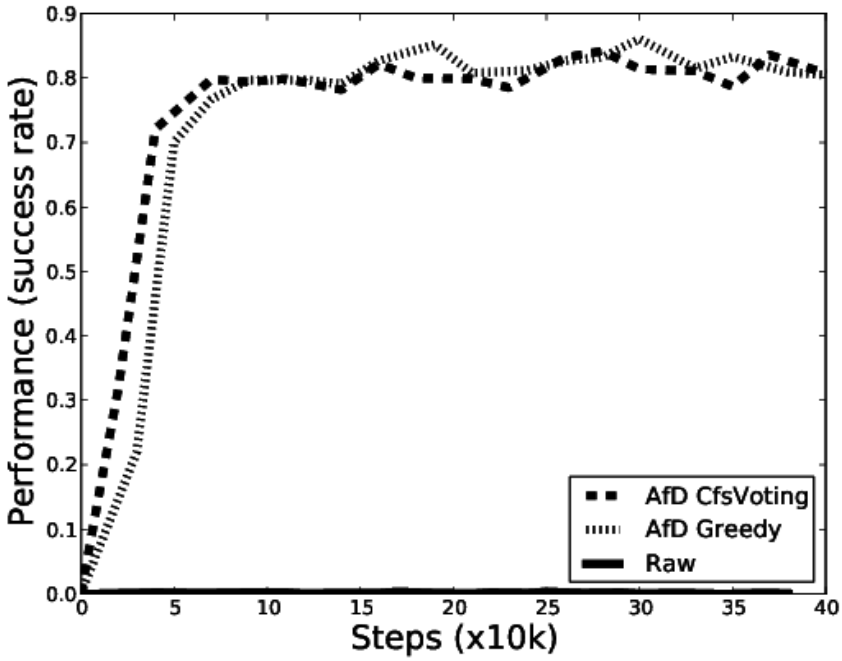


Figure 9: Performance (% of games won) of AfD and learning in the raw state space in the Frogger domain.

were also useful: cells in the three rows contiguous to the frog and others that allowed the frog to line up with a goal cell when in a safe row.

We also compared the performance of AfD to that of applying RL directly in the raw feature space. Figure 9 shows that working in the large raw state space did not achieve significant learning: states are rarely visited for a second time, retarding learning. Additionally, memory consumption grows linearly as a function of the number of steps taken. In our experiments, the raw method had consumed 19GB of memory before it had to be killed. At that point, it had taken almost 1.7 million steps, but the success rate was still below 0.2%. By contrast, AfD was performing better within a thousand steps (the success rate is lower than in Table 2 because exploration was not disabled). This dramatic difference reflects the reduction in state space, and the corresponding exponential reduction in computational cost. Figure 10 illustrates the difference in state-space size by showing the growth of policy sizes (the

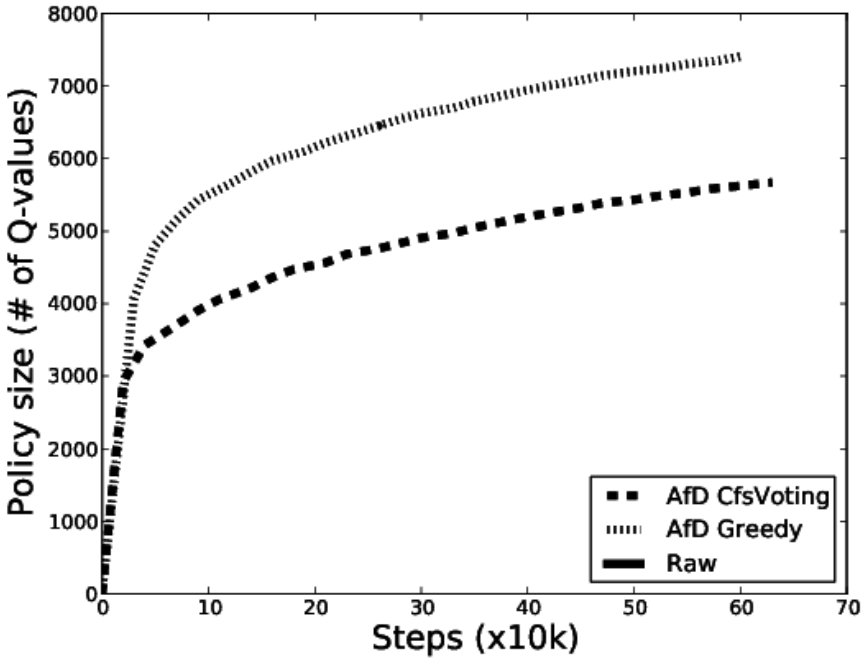


Figure 10: Policy size vs. number of steps of AfD and learning in the raw state space in the Frogger domain. The raw line is not visible because it matches the vertical axis.

raw method is aligned with the y-axis). Policy sizes for the AfD methods begin to level off immediately.

Learning in the raw domain, the size of the representation of the policy grows linearly with the number of steps, since at each time step the algorithm visits a previously unseen state for which a Q-value must be remembered. It cannot be seen in Figure 10 because the line for learning without demonstrations is aligned with the y-axis, but after 1.7 million steps, 92% of the steps were a previously unseen state-action.

6.5.3 Results on hierarchical domains

Using the hierarchical Panda domains described in Section 6.5.1.3, we compare ADA with reinforcement learning using Sarsa(λ), learning from demonstration using a C4.5 [61] decision tree, and abstraction from demonstration. We first discuss RL

Table 4: LfD results over 10 thousand episodes, using a C4.5 decision tree. Average steps computed only over successful episodes.

Domain/Player	Episodes	Success rate	Avg. steps
Sequential	100	0.28%	241.21
	200	0.44%	227.79
	400	0.82%	242.83
Rainbow/A	100	0.81%	1543.84
	200	2.76%	1650.37
Rainbow/B	100	0.27%	1518.70
	200	0.27%	1994.92
	300	0.73%	1901.51

and LfD, then our algorithm, and finally we compare the abstractions that our ADA and AfD find.

6.5.3.1 Reinforcement learning using Sarsa

For Sarsa, we discretized the continuous values into 64 bins. The results were poor in these domains because of the high dimensionality of the problems. The simpler domain, PandaSequential, still has 64^6 possible states, for a total of about 343 billion Q-values. To ensure our implementation of the algorithm was correct and find its limits, we tested it with simplified versions of the game, with only one and two balls. The results are shown in Figure 11. We can see that Sarsa performed reasonably well for the case of only one ball (4096 states), but no longer did so for the two-ball game (16.8 million states), even though we let the algorithm run for 8 days on a modern computer, this is, 2000 times longer than it took for the one-ball policy to converge. The policy performance keeps improving, but at an extremely slow rate. Therefore, Sarsa is not an effective option for these domains.

6.5.3.2 Learning from demonstration using C4.5

To compare with the performance of traditional LfD techniques, we trained a C4.5 [61] decision tree with the demonstrations captured from human players, in a purely supervised learning fashion. Table 4 shows that LfD also performed poorly. The best

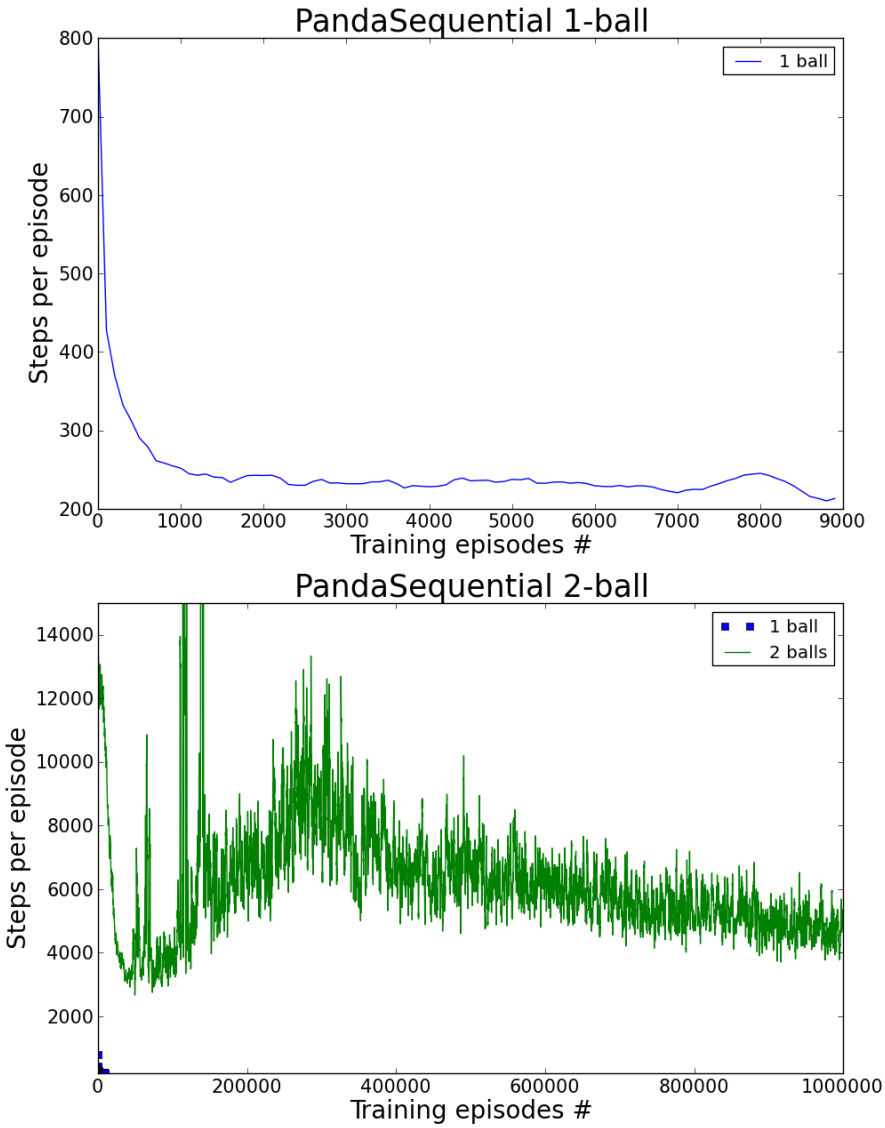


Figure 11: Results using the Sarsa(λ) algorithm on a simplified version of the domain with only one or two balls. One extra ball decreases performance by two orders of magnitude, even if the algorithm trains 2000 times longer.

Table 5: Comparison of human performance, ADA, and ADA+RL, measured in number of steps to task completion, averaged over 10 thousand episodes.

Domain/Player	Episodes	Human	ADA	ADA+RL
Sequential	100	322.67	360.75	267.11
	200	318.71	360.11	
	400	311.30	335.92	
Rainbow/A	100	525.33	-	-
	200	504.71	626.27	466.59
Rainbow/B	100	540.03	596.46	
	200	536.43	593.45	
	300	533.56	583.34	

result we obtained, using all available demonstrations, was a policy that would reach the goal state in less than 3% of the episodes.[2]

6.5.3.3 Automatic decomposition and abstraction from demonstration

To use ADA in our domains, we discretized the continuous values into 64 bins (same as for RL/Sarsa), used $\epsilon = 0.1$, and considered as candidate boundaries every possible threshold on every feature of the domain. ADA was much more effective than the other methods on both domains, and led to near-optimal polices that succeeded on every episode. Table 5 shows that we obtained policies comparable to those of the human teacher. Even though the average number of steps is slightly higher than for the LfD policy in the PandaSequential domain, this is averaged over all episodes, while the number for LfD is only averaged over the small percentage of episodes that LfD is able to resolve.

The success of ADA, compared with LfD and Sarsa, is due to its finding the right decomposition of the domains. For both domains, the algorithm builds an abstraction that focuses only on the angle between the agent and the next ball to be picked up. Which ball is the target depends on what balls are present for the PandaSequential domain, and on the current color the agent is targeting for the RainbowPanda domain.

[2]It should not be a surprise that sometimes, with more samples, the number of average steps on successful episodes increases. This is due to the policy being able to deal with more difficult episodes (remember that the initial placement of the balls is random) that require more steps to complete.

The algorithm was able to identify the right boundary on each domain. It was a surprise that only the angle, and not the distance to the ball, was necessary, but it is easy to see that a satisficing policy can be found using only the angle: rotate until the ball is in front of the agent and then go forward. In fact, this was what the human players were doing, except in the rare case where the ball of interest was right behind the agent; since the agent moves faster forward than backward, it was usually not worth moving backwards.

Only one case in Table 5 did not produce the abstraction described above. Rainbow/B-100 episodes did not find any abstraction. This was due to m_{ss} being higher than a third of the total number of samples; therefore, it could not find any of the three subtasks, one per color and roughly of the same size, that were found in the other cases. We tested a lower value for ϵ and in that case the usual abstraction was found.

The same table shows results for $ADA + RL$, applying policy search on top of the policy found by ADA. The abstraction built by ADA may prevent boot-strapping algorithms such as Sarsa or Q-learning from converging, but with only 192 states in the abstraction and a good starting policy, we can use direct policy search methods. We could obtain good results by just iteratively changing the policy of each state and evaluating the effect on performance using rollouts.

Using this additional policy improvement step, we can find policies that are better than those demonstrated by the human teachers. The policies found were better than those demonstrated in three ways. First, the preferred action for states that were rarely visited was sometimes incorrect in the ADA policy, because there were not enough samples in the demonstrations. ADA+RL could find the best action for these uncommon states. Second, human players would make the agent turn to face the target ball and then move forward when the relative angle to the ball was less than 15 degrees. ADA+RL found it was more efficient to turn until the angle to the ball was less than three degrees, and only then move forward. Third, ADA policies assign

some probability to each action depending how often it is taken in the demonstrations for a particular state. ADA+RL can identify which actions were not appropriate for the state and never execute them, even if they appear in the demonstrations, perhaps because of distractions or errors from the teacher. In short, the policy found by ADA+RL was a *more precise* and *less noisy* version of the policy derived directly from the demonstrations.

6.5.3.4 Abstraction from demonstration

Finally, we tried AfD in the domains, using the abstraction algorithm described in Section 6.3.3 for the whole state space. In the PandaSequential domain, AfD would identify the position of the first ball as the only useful feature. This abstraction leads to a policy that can find the first ball quickly, but can only perform a random walk to find the other two balls. The large difference in mutual information between each ball position and the action is due to the fact that while the first ball is present, its position is significant for the policy; however, the second ball is significant for the policy only half of the time it is present, and the third ball only a third of the time it is present.

Regarding AfD for the RainbowPanda domain, because the active color at each moment is chosen at random, the mutual information measures between each ball's relative position and the action are similar. In this case, AfD is able to identify the true relevant features, *i.e.,* the relative position to the closest ball of each color. Due to the nature of AfD abstraction, we could not use bootstrapping algorithms such as Sarsa, and $64^3 = 262144$ states are too many for our naive policy search, so we tried to obtain a policy using Monte-Carlo methods. Unfortunately, these are known to be much slower to converge than Sarsa. Even after experimenting with various exploration parameters, we could not reach a policy better than a random walk.

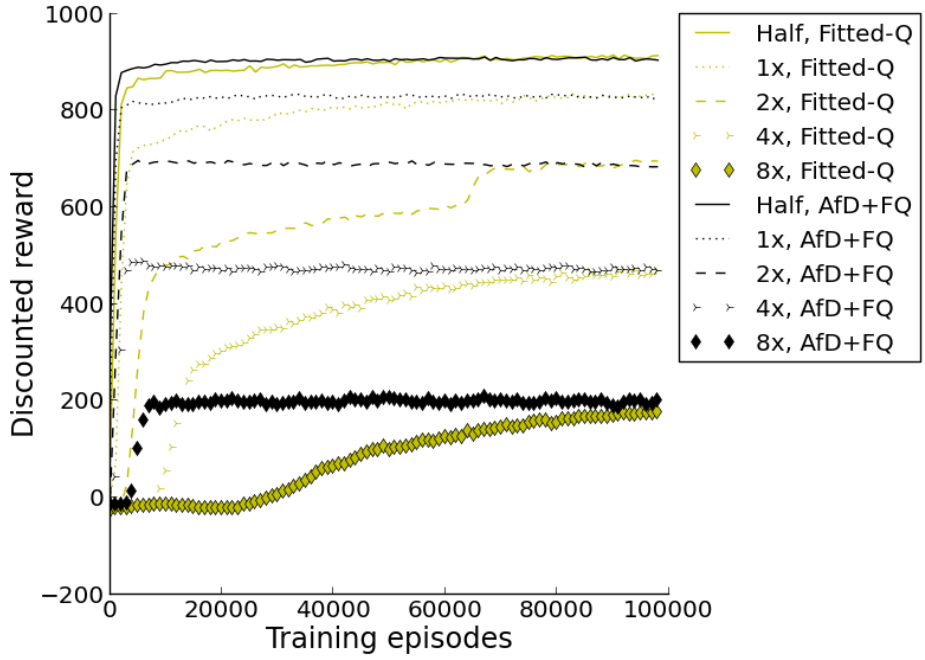
We can thus conclude that for complex domains that can be decomposed in different subtasks, ADA can find policies better than those demonstrated by humans, while traditional LfD, RL, and AfD cannot find policies significantly better than a random walk.

6.5.4 Function approximation

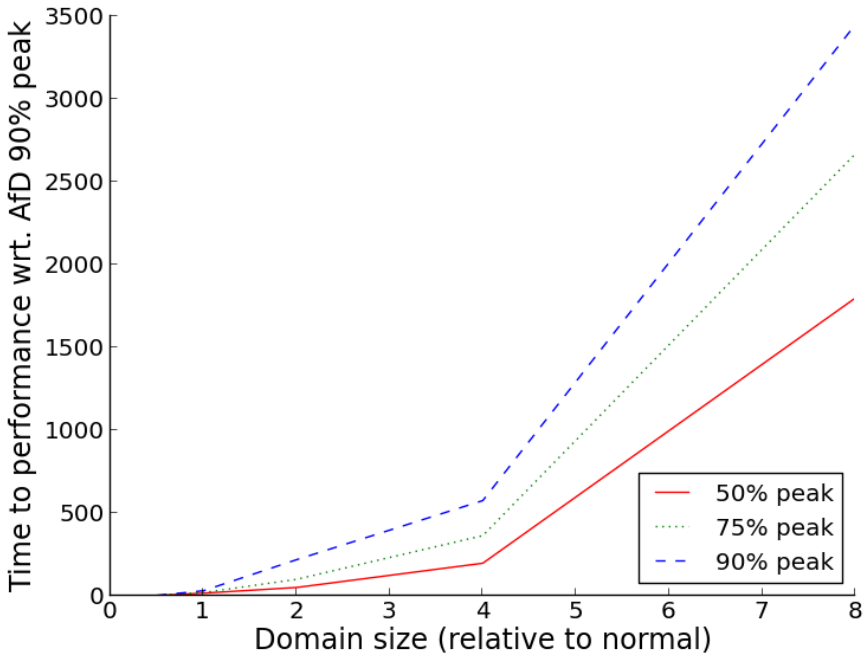
This section experimentally confirms that attention focus still provides significant speedups in high-dimensional domains when used in combination with previous function approximation algorithms. We tried the combination of both AfD and ADA with some of the most popular function approximation algorithms, namely, fitted Q-learning and LSPI, described in Chapter 3. The abstractions we use are the same as those found in our tabular-based experiments.

To measure how the size of the problem affects the speed-up provided by AfD, we experimented with different sizes of the Frogger domain. We tried variations with half, two, four, and eight times the number of car lanes on the domain. For hierarchical domains, we also experimented varying PandaSequential from one to three balls to see the effects of ADA with respect to the dimensionality of the problem.

6.5.4.1 *AfD with fitted Q-learning*



(a) Reward vs. episodes across domain sizes.



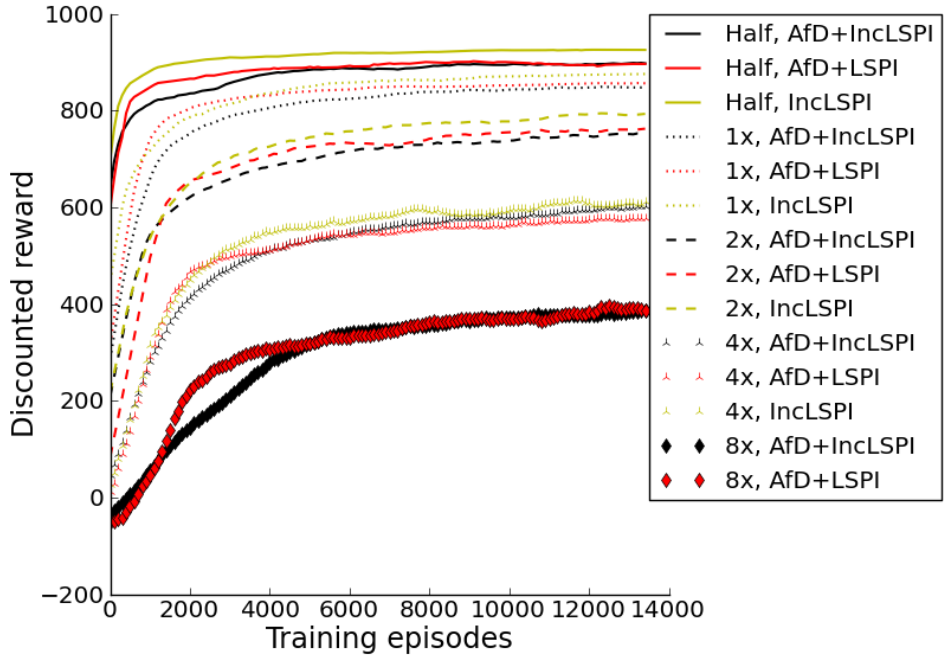
(b) Speed-up provided by AfD vs. domain size.

Figure 12: Results with fitted Q-learning on Frogger domain, averaged over 10 runs.

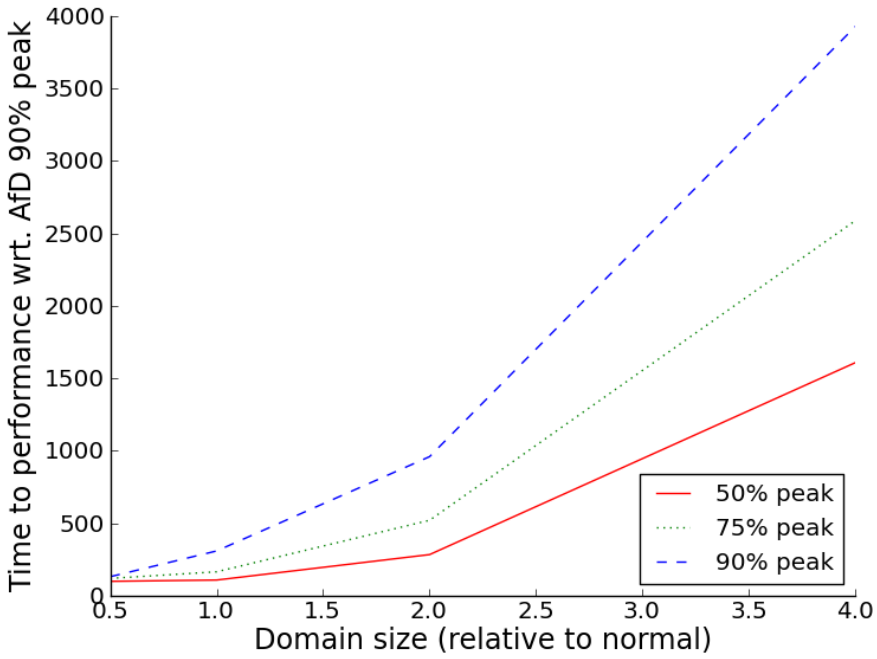
Figure 12(a) shows the average discounted rewards vs. number of training episodes for fitted Q-learning, with and without the AfD abstraction (AfD+FQ and fitted Q). The peak performance is lower for bigger domain sizes, because the final reward is fixed but discounted by the number of steps, and it takes more steps to reach the goal on larger domains. Even though AfD+FQ uses fewer state features to approximate a policy, its final performance does not degrade with respect to the non-AfD algorithm. Learning on the original state space is much slower, and the difference grows with the size of the domain. The performance of AfD+FQ also deteriorates, but most of that effect is because the average time needed to find one of the goals by chance grows with the square of the domain size, since the agent is only avoiding death and performing a random walk in early learning. AfD+FQ converges, with a steep performance slope, to the optimal policy, regardless of the size of the domain. Whereas for non-AfD fitted Q-learning, the convergence slope degrades significantly with the problem size.

Figure 12(b) shows the time it takes fitted Q-learning on the original state space to reach a level of performance comparable to AfD+FQ with respect to the size of the domain. The differences here are larger because, besides requiring fewer episodes for convergence, AfD fitted Q-learning requires fewer steps per episode and each step is faster (fitted Q-learning step complexity is linear in the number of features). The y-axis shows the computational time that the non-AfD algorithm needed to achieve a given fraction of the peak performance, divided by time required by the AfD version to reach 90% of the peak performance. For the largest domain, the non-AfD algorithm needed 3500 times longer to reach the performance of AfD+FQ, approximately 10 days instead of 4 minutes.

6.5.4.2 AfD with LSPI



(a) Reward vs. episodes accross domain sizes.



(b) Speed-up provided by AfD vs. domain size.

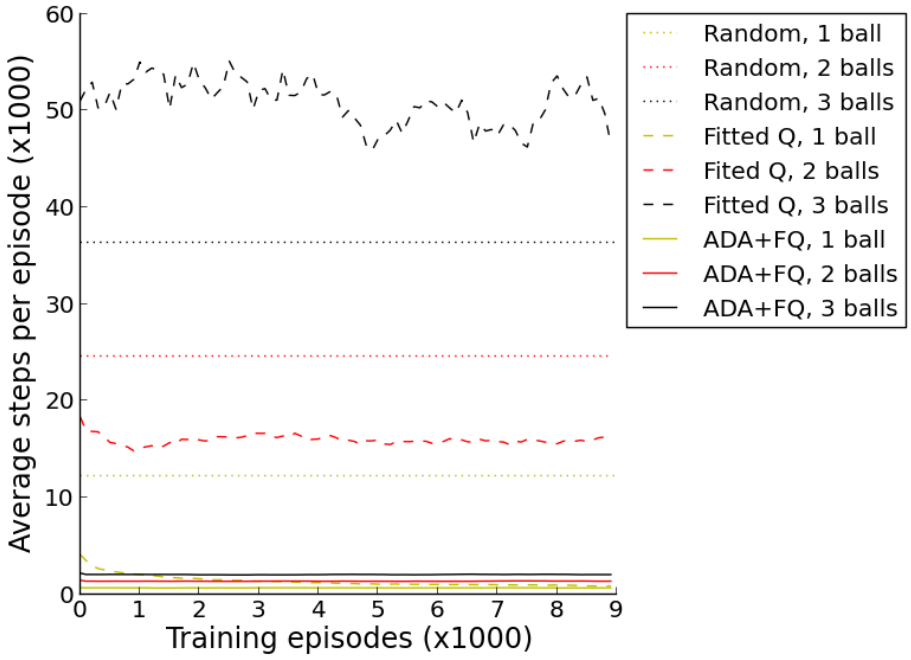
Figure 13: Results with LSPI on the Frogger domain, averaged over 10 runs.

Figure 13 shows the results for LSPI, with a linearly decaying learning rate $\alpha = 0.001$ for IncLSPI. This experiment used both LSPI and IncLSPI combined with AfD, obtaining similar results. Due to the high dimensionality of the problem, it was not possible to use LSPI with the original state space, even for the smallest version of the domain.

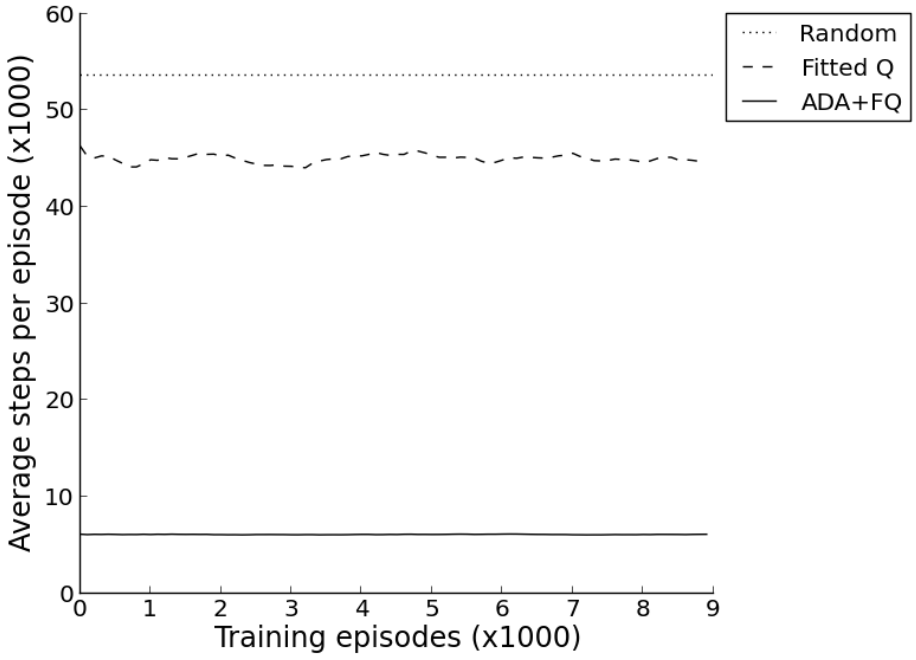
We could expect the non-AfD version to perform slightly better since it has access to the whole state space, but we were surprised to see that the convergence speed in samples was the same with or without attention focus. We believe this is due to the simplicity of the Frogger transition model, which allows model-based LSPI to learn in parallel how the state features are connected. Even though attention focus does not make a difference in the number of samples required for convergence, it still offers significant polynomial speed-ups, as can be seen in Figure 13(b). The more efficient incremental implementation, IncLSPI, still has $O(n^2)$ complexity per sample with respect to the number of features. Non-AfD IncLSPI on the 4x domain took three weeks to complete 15000 episodes, while the AfD version ran the same number of episodes with similar performance in just 87 minutes. We were unable to run non-AfD LSPI on the 8x version: the raw state space has a total of 55000 features, which made it impossible to run a single episode on a 16-core 12GB RAM machine. From this experiment, we conclude that by drastically reducing the number of features, attention focus makes LSPI applicable to larger domains.

We also tried using LARS-TD in this domain to compare feature selection through regularization and our feature selection with human demonstrations. With a regularization criteria of 0.001, LARS-TD selected 300 non-zero features after the first iteration. This is many more than the nine features suggested by AfD. LARS-TD time complexity of $O(mnk^3)$ outweighs the computational gains provided by the feature selection, so the algorithm could not resolve even the smallest version of the domain in weeks.

6.5.4.3 ADA with fitted Q-learning



(a) Average steps per episode for PandaSequential.



(b) Average steps per episode for RainbowPanda.

Figure 14: Results with fitted Q-learning on Panda domains, averaged over 10 runs.

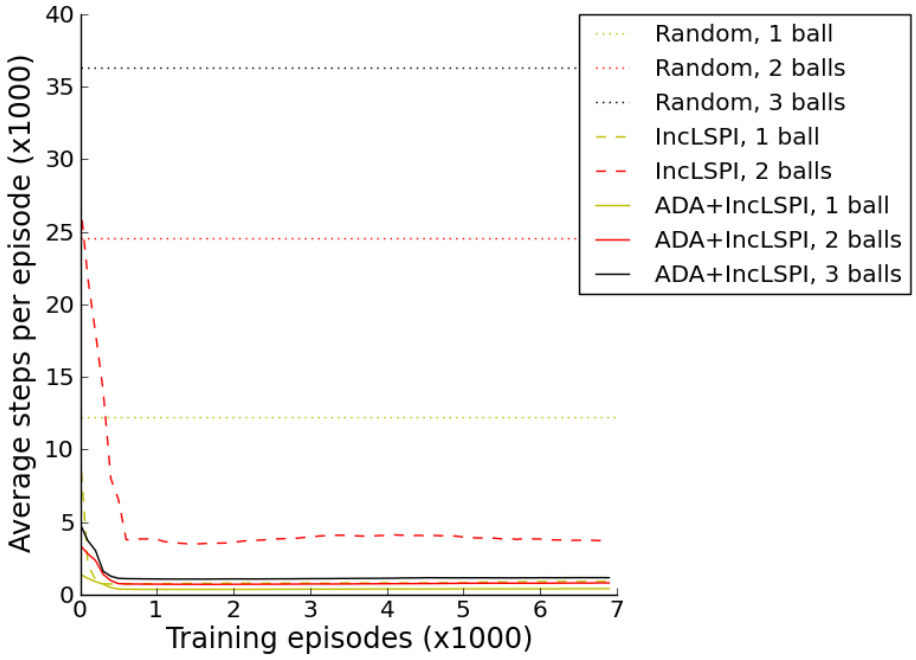
We now compare linear fitted Q-learning on the original state space and on the ADA abstraction in the PandaSequential and RainbowPanda domains. For PandaSequential we compare versions with one, two, and three balls to study the scalability of the algorithms. The results are shown in Figure 14(a). For clarity, these graphs show the average number of steps that it took to complete the task, so a lower value means better performance. With only one ball there is only one subtask; therefore, the ADA abstraction is equivalent to AfD. The convergence of the non-ADA version is extremely slow and takes almost 10000 episodes, while the ADA version needs only a few episodes. This is an interesting result because the difference between both algorithms in the one-ball case is whether they take into account only the distance to the ball or both distance and angle to the ball. Using both parameters could lead to slightly better policies, but we confirmed that, even after 50000 episodes, the policy of the non-ADA algorithm was not better than ADA.

The two-ball and three-ball cases of PandaSequential are also interesting. We expected that the non-ADA version would be able to learn a good policy for this domain. Because the balls are always picked in the same order, it is possible to express a near-optimal policy as a linear combination of the state features: replicate the weights for the features of the first ball scaled down for the subsequent balls, so that the weights associated with the next ball to pick up always dominate; however, fitted-Q was not able to find these policies and performed just a bit better than the random policy for two balls and even worse than the random policy for three balls. The ADA version of the algorithm converges equally fast in number of episodes, regardless of number of balls, and the length of the episodes grows linearly with the number of balls, as expected for a near-optimal policy.

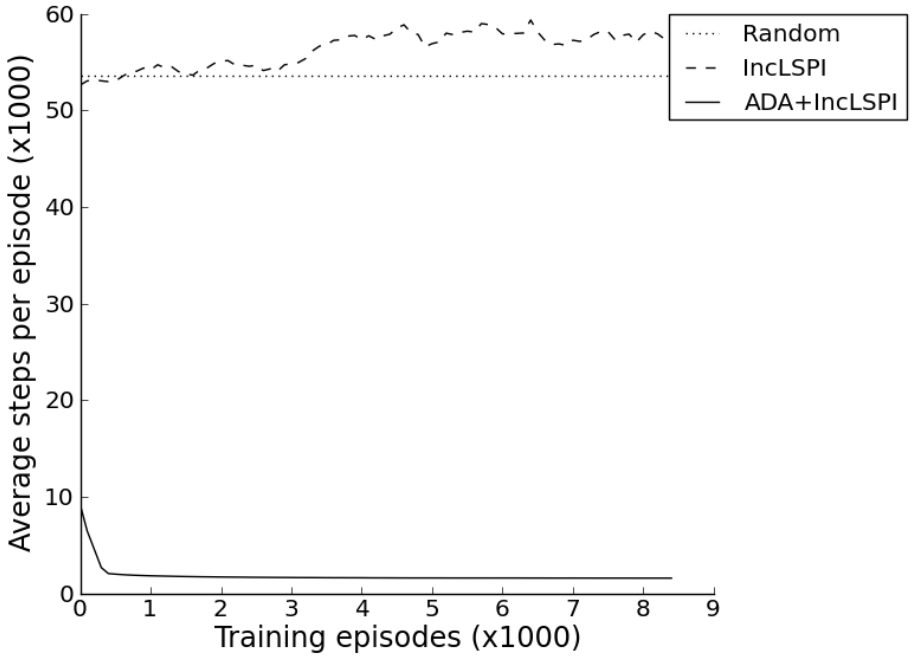
We correctly expected poor results for RainbowPanda, because in this domain it is not possible to represent a global policy using only a linear combination of features. We could represent a global policy only if we add features like the Cartesian product of

the position of each ball and the active color, but that entails manually coding domain-specific information, which is what attention focus tries to avoid. Figure 14(b) shows that, unable to represent the policy, the non-ADA version of the algorithm performs poorly while the ADA version converges quickly to a near-optimal policy.

6.5.4.4 ADA with LSPI



(a) Average steps per episode for PandaSequential.



(b) Average steps per episode for RainbowPanda.

Figure 15: Results with LSPI on Panda domains, averaged over 10 runs.

Figures 15(a) and 15(b) show an effect on LSPI similar to the one on fitted Q-learning. ADA expands the hypotheses space of LSPI (incLSPI with initial $\alpha = 10^{-8}$) so that it can find near-optimal policies for more complex domains. For PandaSequential, the non-ADA policy for two balls is three times quicker than the policy found with non-ADA fitted Q-learning. However the three-ball policy for non-ADA LSPI (not shown in the figure) is much worse, oscillating between 120 and 160 thousand steps, more than four times slower than the random policy.

6.6 Discussion

Direct LfD seeks to approximate the mapping from states to actions; however, in reality, the teacher's policy is not necessarily deterministic or stationary. Different teachers may also teach different but equally valid policies. Thus, for any given state, there may be a set of equally appropriate actions; hence, we are in a multi-label supervised learning setting [85], but the data is not multi-label.

For any single action, we see examples of when it should be used, but never examples of when it should not be. This problem stems from the fundamental fact that when a teacher shows an appropriate action for a given state, they are not necessarily indicating that other actions are inappropriate. This produces a situation commonly referred to as "positive and unlabeled data" [45]. One of the approaches to deal with positive and unlabeled data is to assume that observed positive examples are chosen randomly from the set of all positive examples [22]. Unfortunately, this is not true for human demonstrations. Attention focus approaches avoid this issue by only using the data to identify relevant features. It does not matter how well we can approximate the demonstrated policy, only that a feature has a positive contribution toward the approximation.

One "unfair" advantage of ADA over the other methods is that we must provide it with a set B of candidate boundaries, which is after all a form of domain information.

In principle, we could chose as boundary every possible subset of S, but this would be computationally intractable, so we must explicitly choose the candidate boundaries. This is a small price to pay for the performance gains of the algorithm. As a default choice, axis-aligned boundaries, i.e., thresholds on a single feature, are a compact class that works well across a diverse range of domains. They would work for the Taxi domain [17], which is the typical example of task decomposition in RL, using as a boundary whether the passenger has been picked up or not. If the features are learned from low-level sensing information using unsupervised feature learning techniques, it is likely that one of the generated features will provide adequate thresholds. Additionally, many learning algorithms have similar kinds of bias, *e.g.,* decision trees also consider only thresholds on a single feature, just like ADA in our experimental setup.

A significant limitation of ADA is that it does not consider second-order mutual information relationships, and these can be relevant. For example, we can imagine a domain where the desired action depends on whether two independent random variables have the same value. The mutual information between each variable and the action might be 0, but the mutual information between both variables and the action would account for all the entropy of the action. We have decided to use only first-order mutual information because we believe it is enough to obtain a good decomposition of a wide range of problems, and because the number of samples needed to get an accurate estimate of higher-order relationships is much larger. However, if a large number of demonstrations is available, ADA can be easily extended to use these additional mutual information measures.

One additional advantage of ADA is that it can be used as part of a larger system of **transfer learning**. Once an autonomous agent learns a new task and the subtasks it decomposes into, the subtask policies can be useful for other tasks that may be decomposed in a similar way. For example, an agent might, as part of the policy

improvement step for a specific subtask, try policies of previously learned subtasks that have the same abstraction, maybe after comparing policies and determining that the subtasks are similar. Demonstrating the utility of ADA for transfer learning is an important area of future work.

6.7 Conclusions

Attention focus approaches perform better than policies built using direct LfD, even when LfD is using an order of magnitude more demonstrations. Sample efficiency is one of the key advantages of our algorithms, given that obtaining good human demonstrations is often expensive and time-consuming.

Because of the cost of acquiring human samples, it might be desirable to avoid it altogether and use direct RL algorithms without using demonstrations; however, AfD and ADA achieve significant speed-ups by taking advantage of the exponential savings of dimensionality reduction, and converge to a high performance policy in minutes, while learning without demonstrations did not show improvement over the initial random policy, even after days of computation. This speed-up suggests that, in many domains, even including the time and cost required to acquire the human demonstrations, AfD will be more cost-effective and time-effective than learning without using human demonstrations.

Another advantage of attention focus from demonstration is that its performance is not limited to that of the teacher. AfD and ADA use the reward signal to obtain the best policy that can be expressed in their reduced feature space. This policy, as our results show, can be significantly better than that of the teacher.

Attention focus can also extend the class of sequential decision problems that can be solved with FA algorithms such as fitted Q-learning and LSPI, reducing at the same time the amount of manual feature engineering necessary to adapt the algorithms to new domains. This is achieved through polynomial speed-ups with

respect to the number of state features and extensions of the hypothesis space by task decomposition.

CHAPTER VII

ATTENTION FOCUS WITH A WORLD MODEL

This chapter introduces object-focused Q-learning (OF-Q) [14], our algorithm for attention focus based on a world model. Even though it assumes that the world has certain structure, it does not try to recover the model of the MDP; therefore, it is a model-free algorithm. Because our algorithm is similar to previous approaches proposed for modular reinforcement learning, we start with a brief introduction to modular RL. Our approach can also be compared to OO-MDPs. While our approach is less expressive than OO-MDPs, it does not require specific domain knowledge. Additionally, OO-MDPs represent a model-based approach, while OF-Q is model-free.

7.1 Modular RL

In OF-Q, each object produces its own reward signal, and the algorithm learns an independent Q-function and policy for each object class. This makes our algorithm similar to modular reinforcement learning, even though we have different goals than modular RL. Russell & Zimdars [64] take into account the whole state space for the policy of each module, so they can obtain global optimality, at the expense of not addressing the dimensionality problems that we tackle. Sprague & York [69] use different abstractions of the state space for the different module policies, but because they use the Sarsa algorithm to avoid the so-called *illusion of control*, they can no longer assure local convergence for the policy of each individual module. In OF-Q, as we explained in Section 7.2, we take a different approach to avoid this problem: for each object class, we learn the Q-values for optimal and non-optimal policies and use that information for the global control policy. Because we use Q-learning to learn

class-specific policies, we can assure their convergence. Section 7.6 shows that our control policy performs better than the basic command arbitration of modular RL algorithms.

7.2 Object Focused Q-learning

OF-Q is designed for solving episodic MDPs with the following properties:

- The state space S is defined by a variable number of *independent objects*. These objects are organized into *classes* of objects that behave alike. The object independence assumption will be relaxed in Section 7.5.

- The agent is seen as an object of a specific class, constrained to be instantiated exactly once in each state. Other classes can any number of instances, including none, in any particular state.

- Each object provides its own reward signal and the global reward is the sum of all object rewards.

- Rewards from objects can be positive or negative.[1] The objective is to maximize the sum of discounted rewards, but we see negative rewards as punishments that should be avoided, *e.g.,* being eaten by a ghost in a game of Pacman or shot by a bullet in Space Invaders. This construction can be seen as modeling safety constraints in the policy of the agent.

These are reasonable assumptions in problems that autonomous agents face, *e.g.,* a robot that must transport items between two locations while recharging its batteries when needed, keeping its physical integrity, and not harming humans in its surroundings.

[1]This convention is not necessary for the algorithm. We could have only positive rewards and interpret low values as either punishments or just poor rewards; however, to make the explanation of the algorithm more straightforward, we will keep the convention of considering negative rewards as punishments.

For our experiments, we have chosen two domains in the mold of classic videogames, which detail in Section 7.6.1.1. In these domains, an object representation can be easily constructed from the screen state using off-the-shelf vision algorithms. Also, because objects typically provide rewards when they come in contact with the agent, it is also possible to automatically learn object rewards by determining which object is responsible for a change in score.

7.2.1 Object focused MDPs

We formalize an OF MDP as

$$M_{\text{OF}} = (S, A, \{P_c\}_{c \in C}, \{R_c\}_{c \in C}, \gamma)\,,$$

where C is the set of object classes in the domain. A state $s \in S$ is a variable-length collection of objects $s = \{o_a, o_1, \ldots, o_k\}$, $k \geq 1$, that always includes the agent object o_a. Each object o can appear and disappear at any time, and has three properties:

- Object class identifier `o.class` $\in C$.
- Object identifier `o.id`, to track the state transitions of each object between time steps.
- Object state `o.state` $= \{f_1, \cdots, f_n\}$, composed of a class-specific number of features.

There is a separate transition model P_c and reward model R_c for each object class c. Each of these models takes into account only the state of the agent object o_a and the state of a single object of the class c. We assume the model is unknown, so the agent must resort to exploration to find a policy.

Our model differs greatly from OO-MDPs, despite working with similar concepts. In OO-MDPs, besides *objects* and *classes*, the designer must provide the learning algorithm with a series of domain-specific *relations*, which are Boolean functions over

the combined features of two object classes that represent significant events in the environment, and *effect types*, which define how the features of a specific object can be modified. Example effect types could be "increment feature by one", "multiply feature by two" or "set feature to zero". Furthermore, there are several restrictions on these effects, for example, for each action and feature, only effects of one specific type can occur. OF-Q does not require this additional information and is therefore better suited to be used by autonomous agents that may have to learn new tasks without designer intervention.

7.2.2 Algorithm overview

7.2.2.1 Q-value estimation

Q-learning is an off-policy learning algorithm, meaning that Q-values for a given policy can be learned while following a different policy. This allows our algorithm to follow any control policy and still use each object o present in the state to update the Q-values of its class `o.class`. For each object class $c \in C$, our algorithm learns Q_c^*, the Q-function for the **optimal policy** π^*, and Q_c^R, the Q-function for the **random policy** π^R. These Q-functions take as parameters the agent state $\texttt{o}_a\texttt{.state}$, the state of a single object o of class c `o.state`, and an action a. For clarity, we will use s_o to refer to the tuple $\texttt{(o}_a\texttt{.state, o.state)}$ and we will omit the class indicator c from Q-function notation when it can be deduced from context.

With a learning rate α, if the agent takes action a when an object o is in state s_o and observes reward r and next state s'_o, the Q-value estimate for the optimal policy of class *o.class* is updated with the standard Q-learning update in (6), adapted for a specific object:

$$\hat{Q}^*(s_o, a) = (1 - \alpha)\, \hat{Q}^*(s_o, a) + \alpha \left(r + \gamma \max_{a' \in A} \hat{Q}^*(s'_o, a') \right), \tag{15}$$

where the hat denotes that this is an estimate of the true Q^*.

Using the same sample, the Q-value estimate for the random policy of class *o.class*

is updated with

$$\hat{Q}^R(s_o, a) = (1 - \alpha)\, \hat{Q}^R(s_o, a) + \alpha \left(r + \gamma \frac{\sum_{a' \in A} \hat{Q}^R(s'_o, a')}{|A|} \right). \tag{16}$$

7.2.2.2 Control policy

The control policy that we use is simple. We first decide $\mathbb{A}$, the set of actions that are safe:

$$\mathbb{A} = \{a \in A | \forall o \in s, \hat{Q}^R\left(s_o, a\right) > \tau_{\texttt{o.class}}\}, \tag{17}$$

where $\tau_{\text{o.class}}$ is a per-class dynamic threshold obtained as described in Section 7.2.3. The set of all thresholds is $T = \{\tau_c\}_{c \in C}$. The control policy then picks the action $a \in \mathbb{A}$ ($a \in A$ if $\mathbb{A} = \emptyset$) that returns the highest Q-value over all objects,

$$\pi(s)_{\text{OF}} = \arg\max_{a \in \mathbb{A}} \max_{o \in s} \hat{Q}^*\left(s_o, a\right).$$

During learning, we use an ϵ-greedy version of this control policy.

7.2.3 Risk threshold and complete algorithm

Algorithm 3 Object Focused Q-learning algorithm.

for $c \in C$ **do**
  Initialize $\hat{Q}^*_c$
  Initialize $\hat{Q}^R_c$
  Initialize threshold τ_c
end for
$T \leftarrow \{\tau_c\}_{c \in C}$
$\hat{Q}^* \leftarrow \{\hat{Q}^*_c\}_{c \in C}$
$\hat{Q}^R \leftarrow \{\hat{Q}^R_c\}_{c \in C}$
loop
  for $c \in C$ **do**
    $\hat{Q}^*_{\text{control}} \leftarrow \hat{Q}^*$
    $\hat{Q}^R_{\text{control}} \leftarrow \hat{Q}^R$
    candidates $\leftarrow$GetCandidates(T, c)
    stats $\leftarrow$ []
    for $T' \in$ candidates **do**
      candidate_reward $\leftarrow 0$
      for $i \leftarrow 1$ **to** n_evaluations **do**
        episode_reward $\leftarrow 0$
        Observe initial episode state s
        repeat
          $\mathbb{A} \leftarrow$ GetSafeActions$(s, \hat{Q}^R_{\text{control}}, T')$
          $a \leftarrow \epsilon$-greedy$(s, \hat{Q}^*_{\text{control}}, \mathbb{A})$
          Take action a
          Observe new state s and reward r
          Update $\hat{Q}^*$, $\hat{Q}^R$
          Update episode_reward
        until End of episode
        candidate_reward+ = episode_reward
      end for
      stats$[T'] \leftarrow$ candidate_reward
    end for
    $\tau_c \leftarrow$ UpdateThresholds(stats, candidates)
  end for
end loop

The structure of our algorithm is shown in Algorithm 3. We assume that the number of object classes is known in advance, but the algorithm can be extended to handle new classes as they appear. The outer loop determines the safety threshold

set candidates and runs `n_evaluations` episodes with each candidate comparing its performance to that of the current set of thresholds T. At each time step of each episode, our algorithm makes an update to $\hat{Q}^*_{\texttt{o.class}}$ and $\hat{Q}^R_{\texttt{o.class}}$ for each object o in the state s using the update rules in (15) and (16). The policies used for control are only refreshed when the thresholds are updated, so the threshold performance estimation is not affected by changing Q-values. `GetSafeActions` is implemented using (17). The following sections complete the details about threshold initialization and threshold updates.

7.2.3.1 Threshold initialization

To avoid poor actions that result in low rewards, we initialize the threshold for each class as a fraction of $Q_{\min}$, the worst possible Q-value in the domain.[2] For a domain with a minimum reward $r_{\min} < 0$ and discount factor γ, $Q_{\min} = \frac{r_{\min}}{(1-\gamma)}$. In our test domains, negative rewards are always generated by terminal states, so we assume $Q_{\min} = r_{\min}$.

7.2.3.2 Threshold updating

OF-Q thresholds are optimized with a hill climbing algorithm. Starting with a threshold set T, the algorithm iteratively picks each of the available classes $c \in C$ and evaluates two neighbors of T in which the threshold τ_c is slightly increased or decreased. These three candidates are the output of the function `GetCandidates` in Algorithm 3. We have empirically found that a variation factor of 10% from the current threshold works well across different domains. The algorithm runs `n_evaluations` episodes with the current threshold set T and each of the two neighbors to compute the expected reward with each candidate. Then, the threshold τ_c is updated with the value that performs the best.

[2] In the case where only positive rewards are considered, the initial value for the thresholds would be a value in-between $Q_{\min}$ and $Q_{\max}$.

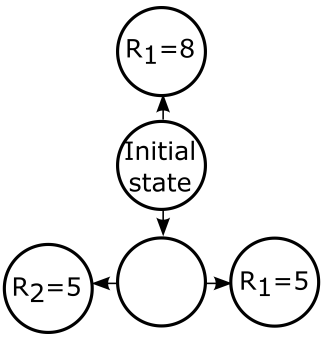
7.3 Benefits of the Arbitration

Our main contribution is a control policy that estimates the risk of ignoring dimensions of the state space using Q-values of non-optimal policies. In this section we explain the benefits of this arbitration.

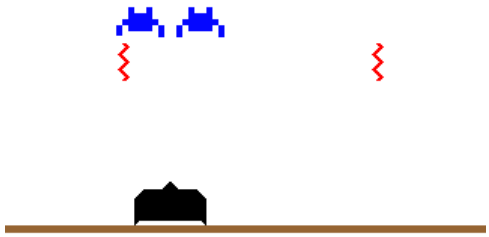
The concept of modules in modular RL literature [69] can be compared to object classes that are always instantiated once and only once in each state; modules never share a policy. This is due, in part, to modular RL aiming to solve a different type of problem than our work; however, modular RL arbitration could be adapted to OF-Q, so we use it as a baseline.

Previous modular RL approaches use a simple arbitration directly derived from the Q-values of the optimal policy of each module. The two usual options are *winner-takes-all* and *greatest-mass*. In *winner-takes-all*, the module that has the highest Q-value for some action in the current state decides the next action to take. In *greatest-mass*, the control policy chooses the action that has the highest sum of Q-values across all modules.

Winner-takes-all is equivalent to our OF-Q control policy with all actions being safe, $\mathbb{A} = A$. The problem with this approach is that it may take an action that is very positive for one object but fatal for the overall reward. In the Space Invaders domain, this control policy would be completely blind to the bombs that the enemies drop, because there will always be an enemy to kill that offers a positive Q-value, while bomb Q-values are always negative.



(a) With these two sources of reward, *greatest-mass* would choose the lower state, expecting a reward of 10. The optimal action is going to the upper state.



(b) For the pessimal Q-values, both bombs are equally dangerous, because they both can possibly hit the ship. Random policy Q-values will identify the closest bomb as a bigger threat.

Figure 16: Arbitration problems.

Greatest-mass is problematic due to the *illusion of control*, represented in Figure 16(a). It does not make sense to sum Q-values from different policies, because Q-values from different modules are defined with respect to different policies, and in subsequent steps we will not be able to follow several policies at once.

In our algorithm, the control policy chooses the action that is acceptable for all the objects in the state and has the highest Q-value for one particular object. To estimate how inconvenient a certain action is with respect to each object, we learn the random policy Q-function Q_c^R for each object class c. Q_c^R is a measure of how dangerous it is to ignore a certain object. As an agent iterates on the risk thresholds, it learns when the risk is too high and a given object should not be ignored.

It would be impossible to measure risk if we were learning only the optimal policy Q-values Q^*. The optimal policy Q-values would not reflect any risk until the risk could not be avoided, because the optimal policy can often evade negative reward at the last moment; however, there are many objects in the state space, and at that *last moment*, a different object in the state may introduce a constraint that prevents the

agent from taking the evasive action. Learning Q-values for the random policy allows us to establish adequate safety margins.

Another option we considered was to measure risk through the *pessimal policy*, *i.e.,* the policy that obtains the lowest possible sum of discounted rewards. This policy can be learned with the update rule

$$\hat{Q}^P(s,a) = (1-\alpha)\,\hat{Q}^P(s,a) + \alpha\left(r + \gamma \min_{a' \in A} \hat{Q}^P(s',a')\right).$$

The pessimal policy offers an upper bound on the risk that an object may pose, which can be useful in certain scenarios; however, this measure of risk is not appropriate for our algorithm. According to the pessimal policy, the two bombs depicted in Figure 16(b) are equally dangerous, because both could possibly hit the agent; however, if we approximate the behavior of the agent while ignoring that bomb as a random walk, it is clear that the bomb to the left poses a higher risk. The Q-values of the random policy correctly convey this information.

In our algorithm, as well as in *winner-takes-all*, the *illusion of control* is not a problem. In Space Invaders, for example, the agent will target the enemy that it can kill soonest while staying safe, *i.e.,* not getting too close to any bomb and not letting any enemy get too close to the bottom of the screen. If the enemy that can be killed the fastest determines the next action, in the next time step the same enemy will be the one that can be killed the soonest, so the agent will keep focusing on that enemy until it is destroyed.

7.4 OF-Q Properties

7.4.1 Class-specific policies

Theorem 1. *Let* $(S, A, \{P_c\}_{c\in C}, \{R_c\}_{c\in C}, \gamma)$ *be an OF MDP.* $\forall c \in C$*, OF-Q Q-function estimates* $\hat{Q}^*_c$*,* $\hat{Q}^R_c$ *converge to the true Q-functions* Q^*_c*,* Q^R_c*.*

Proof. Q-learning converges with probability 1 under the condition of bounded rewards and using, for each update t, a step size α_t such that $\sum_t \alpha_t = \infty$, $\sum_t \alpha_t^2 <$

∞ [79]. Given our independence assumption and that Q-learning is an off-policy algorithm, Q-values can be learned using any exploration policy. If we see the domain as a different MDP executed in parallel for each object class c, the same convergence guarantee applies. Several objects of the same class simultaneously present can be seen as independent episodes of the same MDP. The convergence proof also applies for Q-values of the random policies by changing the maximization by an average over actions. □

Theorem 2. *Let $(S, A, \{P_c\}_{c\in C}, \{R_c\}_{c\in C}, \gamma)$ be an OF MDP and (S, A, P, R, γ) be the equivalent traditional MDP describing the same domain with a single transition and reward function P and R. All OF MDP class-specific Q-functions will converge exponentially faster, in samples and computation, than the MDP Q-function with respect to the number of objects in the domain.*

Proof. Q-learning has a sample complexity $O(n \log n)$ with respect to the number of states in order to obtain a policy arbitrarily close to the optimal one with high probability [35]. Without loss of generality, assuming a domain with m objects of different classes, each with k possible states, the sample complexity of Q-learning on the global MDP would be $O(k^m \log k^m)$, while the sample complexity of each OF-Q class-specific policy would be only $O(k \log k)$.

Regarding computational complexity, assuming the same cost for each Q-update (even though updates on the whole MDP will be more expensive), OF-Q would take m times longer per sample, since it has to update m different policies for each sample. This makes OF-Q computational complexity linear in m, while Q-learning computational complexity is exponential in m, because the sample complexity is already exponential and all samples have at least a unit cost. □

Note that the speed-ups come from considering each object independently, and not from organizing the objects into classes that share a policy. The organization of

objects into classes provides and additional speed-up: if there are l objects of each class on the environment, for example, each sample will provide l updates to each Q-function and the sample complexity would be additionally reduced by a factor of l.

7.4.2 Risk thresholds

Besides deriving the appropriate class-specific Q-functions, OF-Q needs to find an appropriate set of class-specific risk thresholds. As discussed in Section 7.2.3.2, these thresholds are determined with a hill climbing algorithm using the expected reward as the objective function. The episodes used for computing the Q-functions are reused for these optimizations, so if the sample complexity for learning the Q-functions dominates the sample complexity for learning the thresholds, the total OF-Q sample and computational complexity (threshold updating costs are negligible) would be equal to that for learning Q-functions. In this case, our algorithm would provide exponential speed-ups over traditional Q-learning. Experimentally, we observe that in even moderately sized problems such as Space Invaders, the time needed to learn the Q-functions dominates the time needed to converge on thresholds. A definite answer to this question would require complexity analysis of hill climbing, which is outside of the scope of this work.

Given the stochastic nature of our objective function, we can use a hill climbing variant such as PALO [28] to ensure the convergence of thresholds to local optima. To find global optima for the set of thresholds, simulated annealing [37] can be used instead. Unfortunately, PALO and simulated annealing are known to be slower than traditional hill climbing, which has also proved to work well in our experimental domains. Due to these factors, we have chosen a simple hill climbing algorithm for the first implementation of OF-Q.

7.5 Object Dependencies

The assumption that every object is independent is too restrictive for domains where a task must be accomplished by the simultaneous use of several objects. However, inter-object relations are typically sparse, so an object interacts significantly only with a reduced set of other objects. A screwdriver, for example, should be considered in conjunction with the screws it is used with, but can be safely considered independent of all the other objects, *e.g.,* a hammer. This section shows how to adapt OF-Q to these domains with a generic approach that is also applicable to other arbitration functions such as *winner-takes-all* and *greatest-mass*.

7.5.1 Approach overview

We first focus on domains where each class is instantiated at most once. Later we will discuss how our approach can be adapted to the general case of many-instances classes.

Our approach considers every object as an independent reward source. To model the policy for each of these reward sources, we need to observe the object itself, the agent, and an undetermined set of other objects on the domain. We start observing, for each object o_i, only the state of the agent and of the object o_i itself, but we keep track of what the policy performance would be if we also observed the state of each other object in the domain. When we find that observing the state of some other object o_j would increase significantly the policy performance, we start observing that object and we restart the process to include yet another extra object if it can improve performance.

One problem with our approach is that it cannot learn *XOR*-type relationships between objects. If the reward an object o_i provides depends, for example, on two other objects o_j and o_k being in the same state with respect to each other, none of the two objects will seem to influence the performance of the policy for object o_j

if observed without the other. An easy fix is to consider combinations of up to n objects for addition to the policy, but then the same problem appears for the case of combinations of $n+1$ objects if no improvement can be measured when observing any subset. This is a limitation of our approach, but humans are limited in the same way: it is hard for us to discover the effect of many variables if they cannot be isolated to model the individual effects. Additionally, in most real-world problems, if a policy can be improved by observing a number of factors, usually there will also be a partial improvement when a partial list of those factors is observed.

We are tackling the problem of finding the best possible representation for an MDP. Previous work has addressed this problem referring to it as representation switching or abstraction selection [65, 41], but these approaches required either a set of demonstrations (which we assume are not available in OF-Q settings) or rollovers with each possible representation, which is not feasible if the number of possible representations is high.

7.5.2 Domains with single-instance classes

We assume, for clarity, a domain with only one reward source. For the case of many reward sources, we would apply the OF-Q algorithm, adapted so that the policy for each object is independently computed as described in this section.

The domain has an agent object o_a, the reward source object o_{rs}, and an arbitrary number of additional objects $o_1, \ldots, o_n$. At the beginning of the learning process, a list of relevant objects `relevant-objects` is initialized to contain o_a and o_{rs}. We also initialize a Q-function Q_0, indexed by the objects in `relevant-objects` and Q-functions Q_1 to Q_n, with Q_i updated by observing the state of all the objects in `relevant-objects` plus o_i. It is necessary to initialize all Q-functions alike and pessimistically. By pessimistically, we mean that the initial value for each Q-value is the worst possible for the domain, *i.e.,* $\frac{R_{min}}{1-gamma}$ for minimum reward R_{min}.

Algorithm 4 Representation learning for one source of reward and k single-instance classes.

```
relevant-objects ← [o_a, o_rs]
loop
  if relevant-objects changed then
    policies ← []
    Init Q_0 pessimistically, indexed by relevant-objects
    Add Q_0 to policies
    reward[0] ← 0
    for all o_i, 1 ≤ i ≤ k, o_i ∉ relevant-objects do
      Init Q_i pessimistically, indexed by relevant-objects and o_i
      reward[i] ← 0
      Add Q_i to policies
    end for
  end if
  for n ← 1 to n_evaluations do
    Observe initial episode state s
    for all i, Q_i ∈ policies do
      reward[i] += max_a Q_i(s, a)
    end for
    repeat
      Observe state s
      if Exploration step then
        Take random action, update all Q_i ∈ policies
      else
        a = arg max_{i,a'} Q_i(s, a')
        Take action a
        if max_a Q_0(s, a) == max_{i,a} Q_i(s, a) then
          Update all Q_i ∈ policies
        else
          Update Q_i ∈ policies if max_a Q_i(s, a) == max_{a,j} Q_j(s, a)
        end if
      end if
    until End of episode
  end for
  if ∃j, reward[j]/n_evaluations > (1 + ε) reward[0]/n_evaluations then
    Add o_j to relevant-objects
  end if
end loop
```

The procedure, shown in Algorithm 4, is based on Q-learning with ϵ-greedy exploration. For non-exploratory actions, all actions on all Q-functions are checked, and

the one with the highest Q-value is chosen. When Q_0 has the largest Q-value, all Q-functions will be updated. When Q_0 does not have the largest Q-value, only the Q-functions with the highest Q-value for the action chosen (it could be more than one) are updated.

The algorithm keeps an average of the initial state-values (higher Q-value in the start state) for each Q-function, and every m episodes, it checks if there is a Q_i whose average initial state-value is significantly higher than that of Q_0. When such a Q_i is found, object o_i is added to `relevant-objects`, Q_0 is set to Q_i, Q_i is no longer considered and all $Q_j, 1 \leq j \leq n, o_j \notin$ `relevant-objects` are reinitialized so that each one is updated observing all objects in the updated version of `relevant-objects` plus object o_j.

The algorithm works because every Q-function Q_i is a version of Q_0 with an additional distinction. If object o_i does not provide a useful distinction, values in Q_i will have values similar to the corresponding values in Q_0. Some of the values in Q_i may be higher than the corresponding average represented by the value in Q_0, but due to the way we choose actions, those values will get oversampled until they return to the mean. Additionally, since we do pessimistic initialization, all values in Q_i will tend to be lower than the corresponding values in Q_0 simply because they have received less updates. If object o_i provides a useful distinction, some Q-values in Q_i will be higher than their counterparts in Q_0 and some other Q-values will be lower than their Q_0 equivalents, but due to our action selection mechanism, the higher Q-values will spread to the initial states and the algorithm will recognize the useful distinction.

When choosing an action because of a high Q-value not in Q_0, we do not update Q_0 nor the other Q-functions with lower Q-values. Otherwise, we would be overvaluing the states in Q_0 by updating them using information only available when observing additional objects.

7.5.3 Domains with many-instances classes

Our approach to object dependencies is more complicated in domains where there can be many instances of a given class simultaneously present. If reward source o_j is influenced by objects of class S and there are many instances of S in the domain, it is not obvious whether a policy should consider all the instances of S or only some of them, and in the latter case it is still necessary to decide which of the many instances is the one that matters.

Observing all objects quickly becomes computationally intractable, and most often not the right approach in real-world domains. We assume that there is a function ordering all objects of a given class with respect to any other given object. For example, the most natural ordering in a real-world domain would be *distance*. Using this function, we can reduce the many-instances classes domains into single-instance classes domains, where the object classes are *closest-S*, *second-closest-S* and so forth. Without loss of generality, we will use these terms from now on to refer to these classes. The differences with respect to the previous algorithm are as follows:

- The policy $Q_{closest-S}$ receives an update, at each time step, from each object of class S in the domain. However, the received reward would be applied only to the closest object in the domain, deemed responsible for the reward.
- In action selection, $Q_{closest-S}$ will be considered with every object of class S.

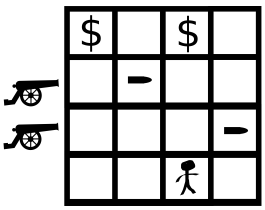
We assume that if any object from class S is connected to a reward obtained from object o_j, it must be the closest one. Our algorithm works well if the closest n must be considered, because it will add them one by one to `relevant-objects`. However, an object from class S that is not the closest in a specific time step, may become the closest after a few time steps. That is why it important to update $Q_{closest-S}$ with all objects of class S in the domain, and why they are all needed for action selection. Otherwise the agent would be short-sighted, and a nearby object of class S would

blind it from taking appropriate action to receive a later reward from a different object of class S that is further away.

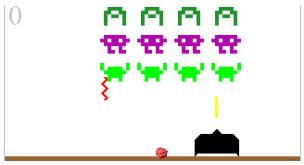
7.6 Experimental Evaluation

7.6.1 Independent objects

7.6.1.1 Domains



(a) Normandy. The agent starts in a random cell in the bottom row and must collect the two rewards randomly placed at the top, avoiding the cannon fire. In our simulations, the grid size is 10x20 with cannons on the third and fourth row.



(b) Space Invaders. In such a high-dimensional state space, traditional reinforcement learning fails to converge in any reasonable amount of time.

Figure 17: OF-Q domains.

We tested OF-Q for independent objects in two different domains. The first one, which we call *Normandy*, is a 10x20 gridworld where an agent starts in a random cell in the bottom row and must collect two prizes randomly placed in the top row. At each time step, the agent can stay in place or move a cell up, down, left, or right. Additionally, there are cannons to the left of rows 3 and 4 that fire bombs that move one cell to the right every time step. Each cannon fires with a probability 0.50 when there is no bomb in its row. The agent receives a reward $r = 100$ for collecting each prize and a negative reward $r = -100$ if it collides with a bomb, and the episodes end when all the rewards are collected or when a bomb hits the agent. Figure 17(a) shows a representation of a reduced version of this domain.

Our second domain is a full game of Space Invaders, shown in Figure 17(b). The

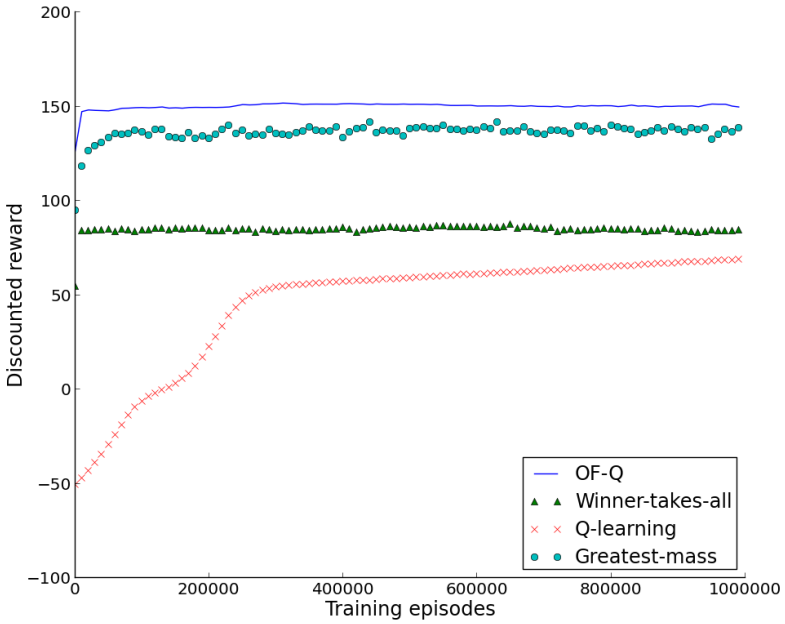
agent controls the ship which, at each time step, may stay in place or move left or right. At the same time, the agent may fire or not, so there are six possible actions. Firing will only work if there is no bullet on the screen at the moment of taking the action. The agent object is thus defined by the x position of the ship and the x and y position of the bullet. The world has two object classes, namely, enemies and bombs, each defined by its x and y position and, in the case of enemies, direction of movement. Initially there are 12 enemies, and each may drop a bomb at each time step with a probability 0.004. The agent receives a positive reward $r = 100$ for destroying an enemy by hitting it with a bullet, and a negative reward $r = -1000$ if a bomb hits the ship or an enemy reaches the bottom of the screen. The game ends when the agent is hit by a bomb, an enemy reaches the bottom, or the agent destroys all enemies.

7.6.1.2 Baselines

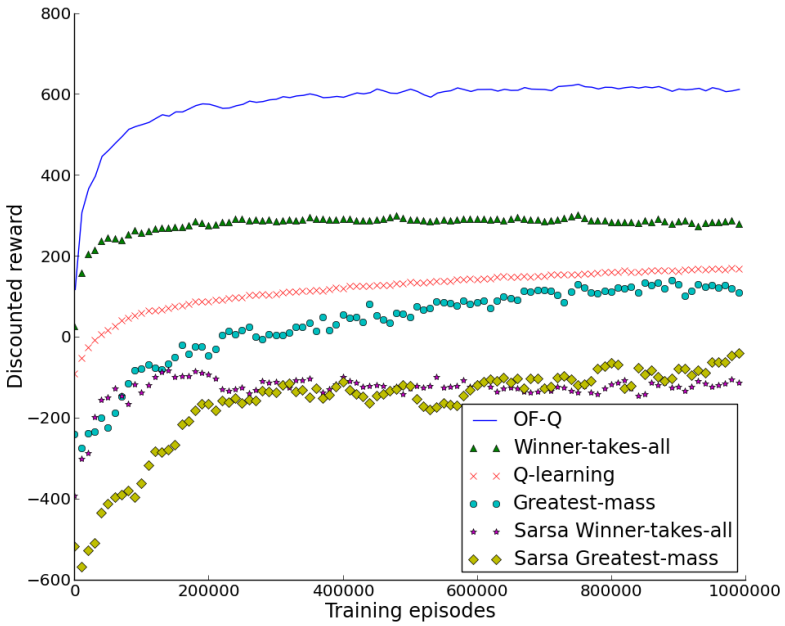
We compare our algorithm with traditional Q-learning (no arbitration), and four other baselines. Two of them are variants of our algorithm using *winner-takes-all* or *greatest-mass* arbitration. These two variants still see the world as composed of objects of different classes and use Q-learning to find an optimal policy for each class, but due to the simpler nature of their arbitration methods, it is not necessary to learn the random-policy Q-values, nor to find a set of risk thresholds. The other two baselines are variants of the algorithms proposed by Sprague & York [69]. These two variants are similar to the previous two, but use Sarsa instead of Q-learning to avoid the illusion of control.

All algorithms and domains use a constant learning rate $\alpha = 0.25$, discount factor $\gamma = 0.99$, and an ϵ-greedy control policy with $\epsilon = 0.05$. For OF-Q, we use 100 evaluation episodes per threshold candidate and an initial threshold of $0.05 \cdot Q_{\min}$ for each class, unless otherwise stated.

7.6.1.3 Results

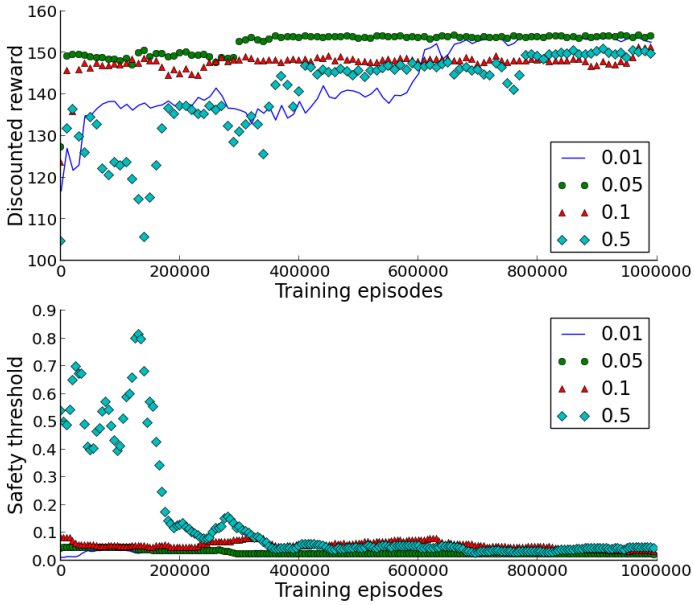


(a) Normandy domain.

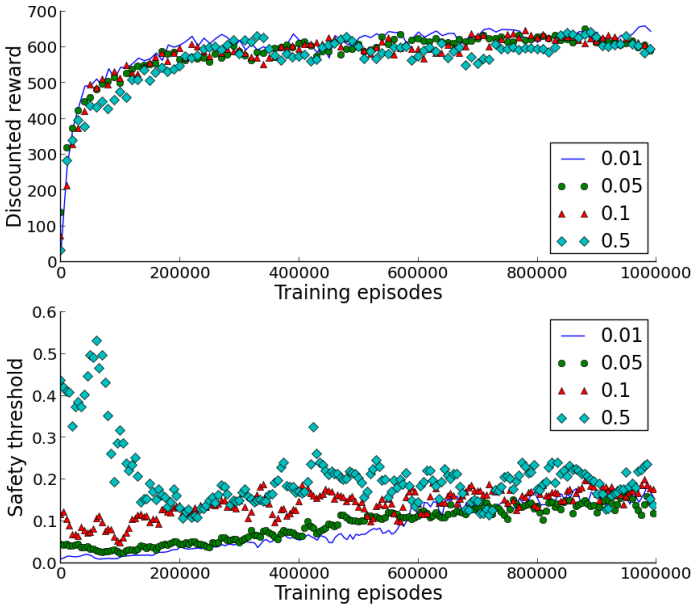


(b) Space Invaders domain

Figure 18: Performance vs. training episodes, averaged over 10 runs.



(a) Normandy domain.



(b) Space Invaders domain

Figure 19: Performance and threshold for bomb objects for single runs with different initial thresholds. The thresholds are expressed as fractions of the minimum Q-value of the domain $Q_{\min}$.

Figure 18 compares the results of our algorithm with baselines on the Normandy and the Space Invaders domain. Our algorithm performs better in both domains, with a larger advantage in the more complex Space Invaders domain. Note that the performance of Q-learning on the Space Invaders domain keeps improving, but at a rather slow rate. After 5 million episodes, the average discounted reward was still below 250. In the Normandy domain, the upward slope of Q-learning is more obvious, and in the 5x5 version of the domain, it did converge within the first million training episodes.

We do not show the results for the Sarsa algorithms on the Normandy domain because the algorithms would not make progress, even after running for days. The algorithm quickly learned how to avoid the cannon fire, but not how to pick the rewards, and therefore the episodes were extremely long. In Space Invaders, these algorithms are the worst performers. These results are not surprising, because Sarsa is an on-policy algorithm and does not offer any convergence guarantees in these settings. If each object class policy was observing the whole state space, the Sarsa policies would converge [64], but we would not get the scalability that OF-Q offers, since it comes from considering a set of low-dimensional policies instead of a high-dimensional one.

The performance of *greatest-mass* varies a lot between each domain, being a close second best option in the Normandy domain and the worst option in Space Invaders. We believe this is due to the *illusion of control* discussed in Section 7.3. The Normandy domain is not affected by this problem, since there are only two sources of positive reward and, at each time step, the agent will go towards the reward that is closest. However, the case for Space Invaders is more complicated. Initially, there are 12 enemies in the domain, all of them a source of reward. The agent can only fire one bullet at a time, meaning that after it fires it cannot fire again until the bullet hits an enemy or disappears at the top of the screen. At each time step, firing is

recommended only by one object, the closest enemy that can be killed; however, the best action for all the other enemies is to not fire yet. Because of the discount factor, the Q-value for the optimal action (fire) for the closest enemy is greater than each of the Q-values for the optimal actions of the rest of the enemies (do not fire yet), but there are more enemy objects whose Q-values recommend not firing. This causes the ship to never fire, because firing at a specific enemy means losing the opportunity to fire on other enemies in the near future.

We ran our algorithm with different initial thresholds to test its robustness. Figure 19 shows that the thresholds for the bombs in both domains converge to the same values, and so does performance, even though there are some oscillations at the beginning when Q-values have not yet converged. Starting with a large value of $0.5 \cdot Q_{\min}$ seems to be a bad option, because even if the policy derived from the Q-values is already correct, the Q-values themselves may still be off by a scaling factor, leading to an ineffective threshold. Nonetheless, even this particularly bad initial value ends up converging and performing as well as the others.

7.6.2 Dependent objects

7.6.2.1 Domains

To test our approach to learning multi-object representations, we implemented a set of domains with an agent on a 5x5 grid-world where there are some sources of reward, a set of objects related with those sources of rewards, and a set of irrelevant objects. The agent has five possible actions: move up, down, right, left, or stay in place. The agent is always identified by its coordinates in the grid, and all domains have three completely irrelevant objects, also identified by their coordinates. All elements are placed at random at the start of each episode. Then, depending on the domain, there are some doors (identified by coordinates) and knobs (identified by coordinates and on-off state) that are placed randomly at each run. The knobs start in the off position and turn on the first time the agent visits the cell where they are located. Figure 20

shows a visual representation of one of the domains.

With these domains, we show that our algorithm learns to observe the elements that are relevant for each source of reward and to ignore the irrelevant objects, building policies that are as complex as needed, but no more complex than that. Next, we detail the features of each domain.

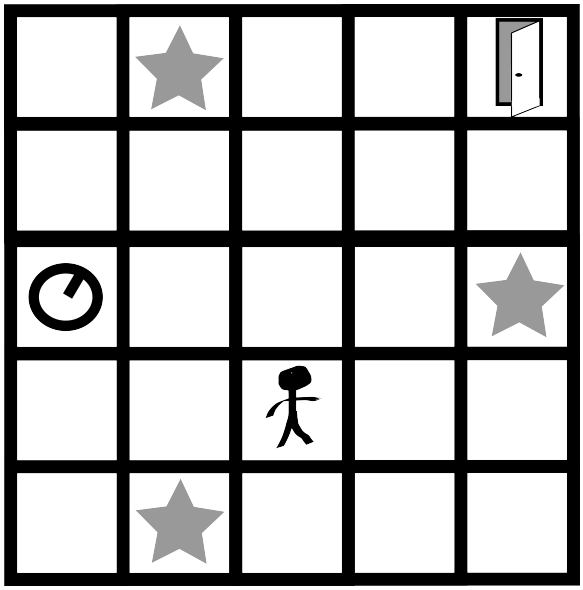


Figure 20: One-Knob domain representation, with one knob, one door, and three irrelevant objects.

One-Knob There is a door, a knob, and three irrelevant objects. The game ends when the agent visits the door with a reward of 100 if the knob is on and -1 if the knob is off. This domain is represented in Figure 20.

Two-Knob There is a door, two knobs, and three irrelevant objects. The game ends when the agent visits the door, with a reward of -1 if both switches are off, 100 if only one switch is on, and 200 if both switches are on.

Create-door There is a door, a knob, and three irrelevant objects. The door is not accessible (it is placed outside of the grid) until the switch is turned on. The game

ends when the agent reaches the door, with a reward of 100.

Direct-reward There is a door, a knob, and three irrelevant objects. The game ends with a reward of 100 when the agent visits the knob, or with a reward of 1 when the agent visits the door.

Combination This domain combines the One-Knob, Create-door, and Direct-reward domains. There are three irrelevant objects and independent knobs and doors for each subtask. In this domain, the doors also have a binary state, so that they start in the active state and they deactivate when in the original domain the episode would have ended. This way, the agent can collect the reward from each domain at most once. The game ends with a reward of 1 when the agent visits an additional door that is located in a fixed position.

7.6.2.2 Baselines

We use two baselines. The first is Q-learning, which builds a single policy considering the whole state space. The second is a variant of OF-Q, which is provided from the start with the right representation for each source of reward. We call this second baseline *fixed representation*. Our learning algorithm is expected to match the performance of the fixed representation baseline, with a slower convergence due to its having to find the right representation for each source of reward.

7.6.2.3 Results

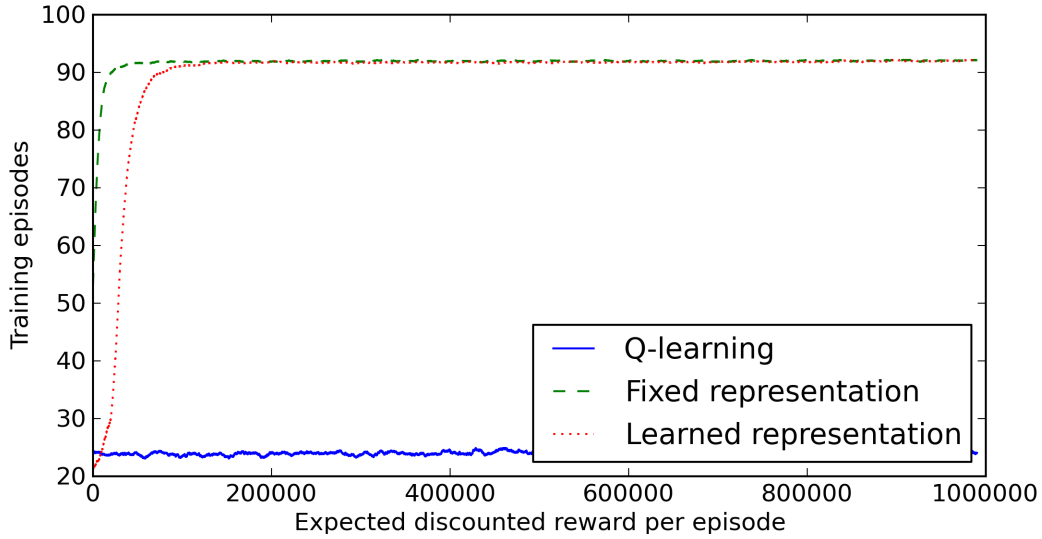


Figure 21: One-Knob domain results, averaged over 10 runs.

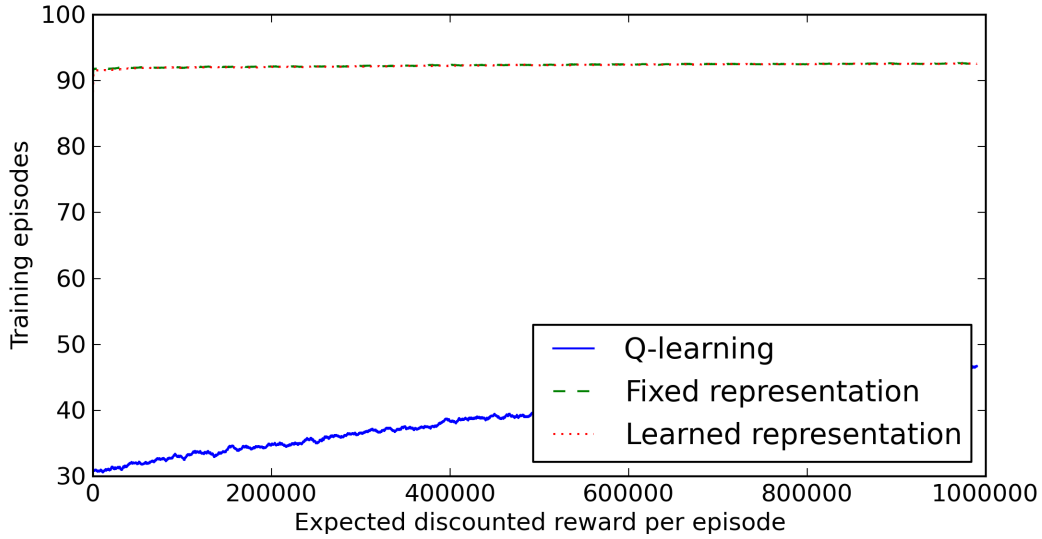


Figure 22: Create-door domain results, averaged over 10 runs.

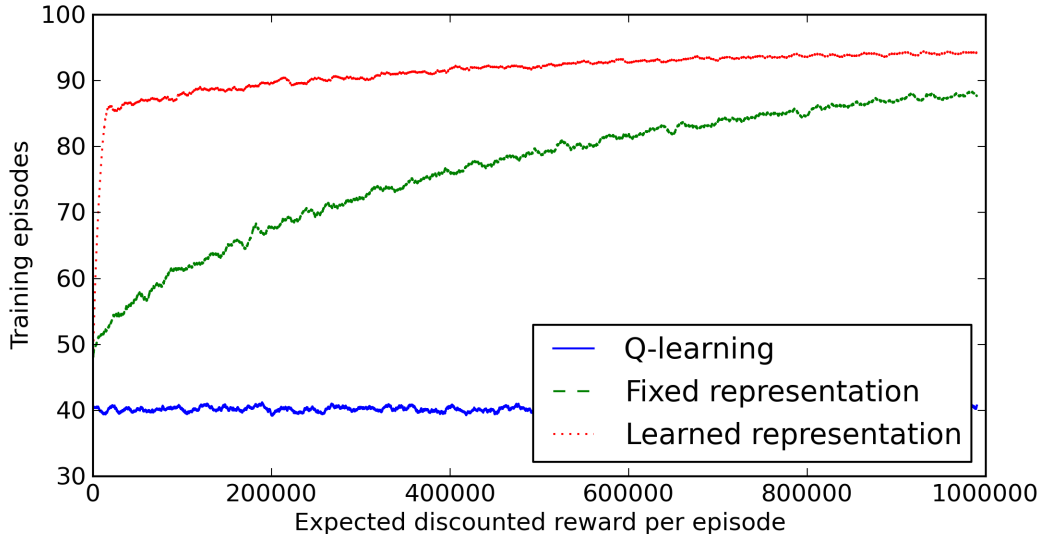


Figure 23: .

Direct-reward domain results, averaged over 10 runs.

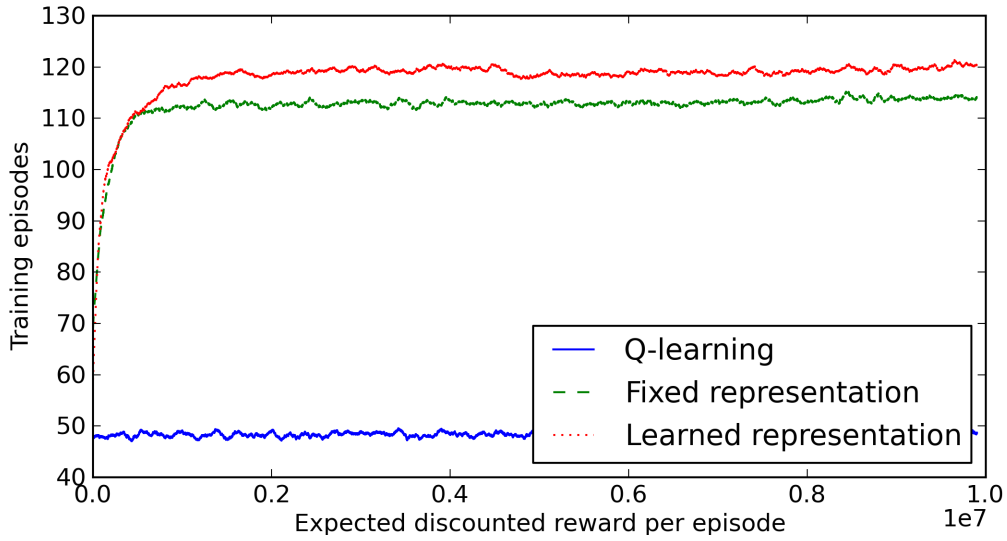


Figure 24: Two-Knob domain results, averaged over 10 runs.

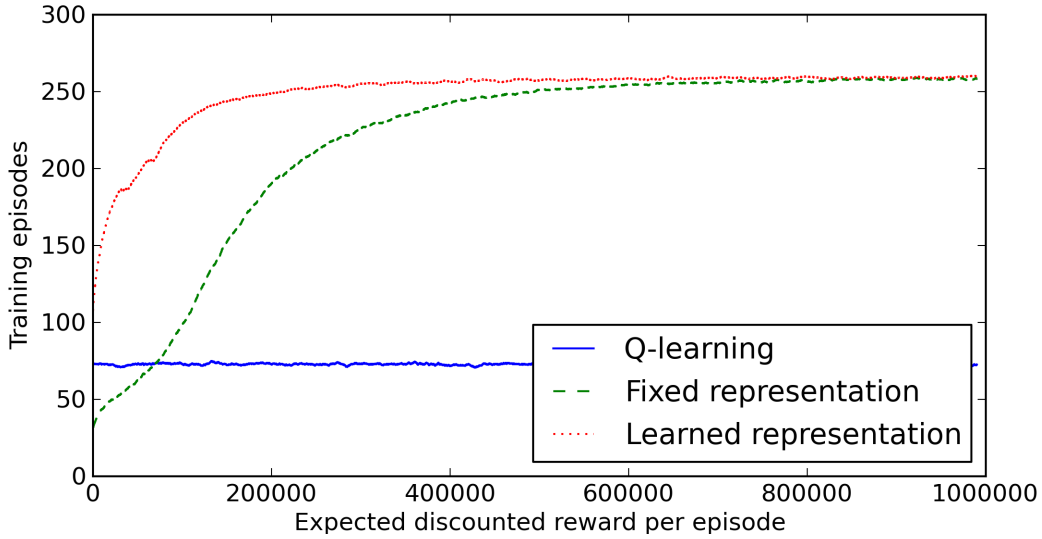


Figure 25: Combination domain results, averaged over 10 runs.

Figures 21 to 25 show the results, where every run of our algorithm found the appropriate representation of the domain, *i.e.,* the one that was provided to the fixed representation algorithm. For the One-Knob and Create-door domains, the representation is learned quickly and the performance of our algorithm catches up quickly with the fixed representation algorithm. The results for Direct-reward, Two-Knob and Combination domains are surprising because our algorithm performance converges faster than the fixed representation baseline. We believe this is caused by the extra exploration that is consequence of the lack of structure in the early stages of the representation learning algorithm. Regarding traditional reinforcement learning, we can see that the domains are too complex for Q-learning to make any significant learning in the first million episodes (or 10 million) episodes.

One interesting effect is that our representation learning algorithm reaches peak performance much earlier than it finds the right representation. This is possible because our algorithm may be effectively using the correct representation when it matters (as it offers better Q-values) even though it has still not decided to use that representation permanently because the higher Q-values did not yet propagate to the

start states. On the Two-knob domain, for example, every run took more than 4.5 million episodes to settle on the final representation (that is why we let this domain run for more episodes than the others), but the performance peaked much sooner, after only one million episodes. This adds robustness to our algorithm, and may be used to speed up the algorithm by adding objects earlier if they are used often for control.

7.7 Discussion

OF-Q works better than previous arbitration algorithms because it is able to learn which actions are acceptable for all reward sources so that it can subsequently safely choose to be greedy. Imagine a domain where there is a big reward with a pit on the way to it, such that trying to get the reward will actually cause the agent to end the episode with a large negative reward. The random policy Q-values with respect to the pit will reflect this possible outcome, even if it takes a rather specific set of actions to reach the pit. The closer in time steps that the agent gets to the pit, the more likely it is that a random policy would lead to the pit and the more negative the Q-values will be. By learning a risk threshold, OF-Q will be able to keep a reasonable safety margin and learn that it is never worth falling into the pit even if there is a big reward ahead. *Winner-takes-all*, on the other hand, would be completely blind to the pit and even *greatest-mass* would still fall in the pit if the positive reward is large enough, even though it will never be reached.

Function approximation can improve RL performance but, in practice, it is often necessary to manually engineer a set of domain-specific features for function approximation to perform well. In many cases, such engineering is a non-trivial task that may essentially solve the abstraction problem for the agent. Because OF-Q breaks the policy needing approximation into simpler components, it will be easier to apply FA to a problem that is decomposed by OF-Q. As we have seen in Section 6.4, the

advantages of attention focus and other function approximation algorithms can be combined, allowing us to tackle domains of higher complexity.

7.8 Conclusions

Object Focused Q-learning (OF-Q) is an attention focus learning algorithm for autonomous agents that offers exponential speed-ups over traditional RL in domains where the state space is defined as a collection of mostly independent objects. OF-Q requires less domain knowledge than earlier relational algorithms; hence it is better suited for use by multipurpose autonomous agents. We proposed a novel arbitration approach based on learning the Q-functions with respect to non-optimal policies to measure the risk that ignoring different dimensions of the state space poses, and we explained why this arbitration performs better than earlier alternatives. Using two videogame domains, including a version of Space Invaders, we showed that our algorithm performed significantly better than previously proposed approaches.

Even though our algorithm was initially designed for domains where all objects are independent, we have shown that it is possible to extend it to domains where it is necessary to consider several objects simultaneously, learning the appropriate representation autonomously. We have experimentally shown that the price paid for this representation learning is small, so our algorithm can still provide significant speed-ups when used for domains where the object independence assumption is not met.

CHAPTER VIII

ATTENTION FOCUS INTERACTIONS

We have introduced two types of attention focus algorithms, one that can be used when demonstrations of the needed task are available, and another one to be used when we can generate or access an object-oriented representation of the domain. This chapter studies how these two approaches can be used together, complementing each other in case both types of information are available.

8.1 Types of Interactions

We will limit ourselves to studying the interactions between an AfD-like (feature selection) algorithm and OF-Q. We believe the overlap between OF-Q and ADA is large enough to make the combination of both methods impractical. An object-oriented representation and reward structure of the domain, like the one OF-Q uses, offers an implicit subtask decomposition. In principle, it might be possible to subdivide each subtask further using an ADA-like approach. To do this, we would have to figure out which part of each demonstration corresponds to each object, which is a difficult problem. Even if we solve this problem, only a small fraction from each demonstration would apply to any given object, and an even smaller fraction would apply to each object-specific subtask. Because of this, the amount of demonstrations needed to find each per-object decomposition makes the approach impractical for domains with a high number of objects.

A more practical approach is to use an AfD-like algorithm to do object-oriented state abstraction as a complementary step in OF-Q. Such abstraction can be applied at three different levels:

Inter-class abstraction To select, among all classes in the domain, the ones that are relevant to finding a good policy. In a typical videogame, for example, it would identify elements that are part of the decoration of the game and do not interact with the agent.

Inter-object abstraction To select, among all instances of a relevant class, which instances are relevant to finding a good policy. For example, in a Mario-like game, an enemy on a distant platform may not be relevant while another one next to the agent will be relevant.

Intra-object abstraction To select, among all features of an object class, which features are relevant to finding a good policy.

The first two types of abstraction offer only moderate speed-ups because their function overlaps with OF-Q. OF-Q already chooses which objects (and therefore classes) are worthy of attention at each time step. By using demonstrations to discard some classes or objects, those objects will not have to be evaluated for their worthiness at each time step, and can be ignored in the learning process. This offers only a linear speed-up, eliminating some iterations in the loop over all objects in OF-Q action selection. Therefore, it is probable that the cost of obtaining demonstrations will not be justified by the benefits that can be obtained. Additionally, inter-object abstraction would need additional domain information in the form of which boundaries or features should be considered in deciding which objects of a given class are important.

The third type, intra-object abstraction, offers more attractive speed-ups. By reducing the number of relevant features in a specific object, we reduce exponentially the size of the Q-table for that object, which leads to exponential speed-ups. For this reason, we focus on this type of abstraction, the most computationally relevant combination of OF-Q and AfD.

8.2 Intra-Object Abstraction

We use intra-object AfD to figure out which features are relevant for each object class so that OF-Q pays attention only to the relevant features of each object. To decide which features are relevant from a demonstration, we use the clustering technique described in Section 6.3.3. However, there are two issues that must be addressed.

The first one is that there might be several objects of the same class present in the domain. This makes evaluation difficult since the demonstrations do not explicitly signal which object was in focus at each time step. Therefore, we do not know which object to use to measure the mutual information between the feature f and the action. Using the feature f of each object of the class simultaneously is impractical because the number and ordering of features needing consideration will vary with the number of objects, and as soon as we contemplate more than a couple of objects, the number of demonstrations needed to obtain reliable mutual information estimations will be quite high. Additionally, even when there is only one instance of the class, the object will be only sometimes in focus. To mitigate this problem, we limit our feature selection to object classes that have only one instance in the domain at least some of the time, and perform the relevant mutual information computations considering only samples in which there is only one instance of the class and the minimum number of other objects.

The second problem is that, as we explain in Section 6.2.2, AfD provides abstractions that may cause bootstrapping algorithms to not work, but OF-Q needs to use Q-learning, a bootstrapping algorithm, so that it can model all object classes simultaneously. To fix this problem, we use the approach to representation switching that we describe in Section 7.5 with just two representations for each object class, the original one (observing all available features) and the one provided by AfD. This way, if AfD provides an abstraction that is harmful for OF-Q, it will be ignored.

A different option that we discarded was to use the approach of Section 6.3.3

for each object, on a per-feature basis. At the beginning, the policy would start observing only the single feature that allows building the best policy, and later it would iteratively add the features that allow the performance of the policy to increase. This may work for the case of independent objects, but in the case of dependent objects, it would interfere with the algorithm that learns the object dependencies: it would not be possible to learn the right dependencies because the necessary features are not considered, and it would not be possible to find the necessary features because the dependencies that need them are unknown.

8.3 Experimental Evaluation

To test the combination of AfD and OF-Q, we modified the Space Invader domain detailed in Section 7.6.1.1 to include two extra features for the enemies (the enemy type and a random integer from 0 to 9) and one extra feature feature for the bombs (one random integer from 0 to 9).

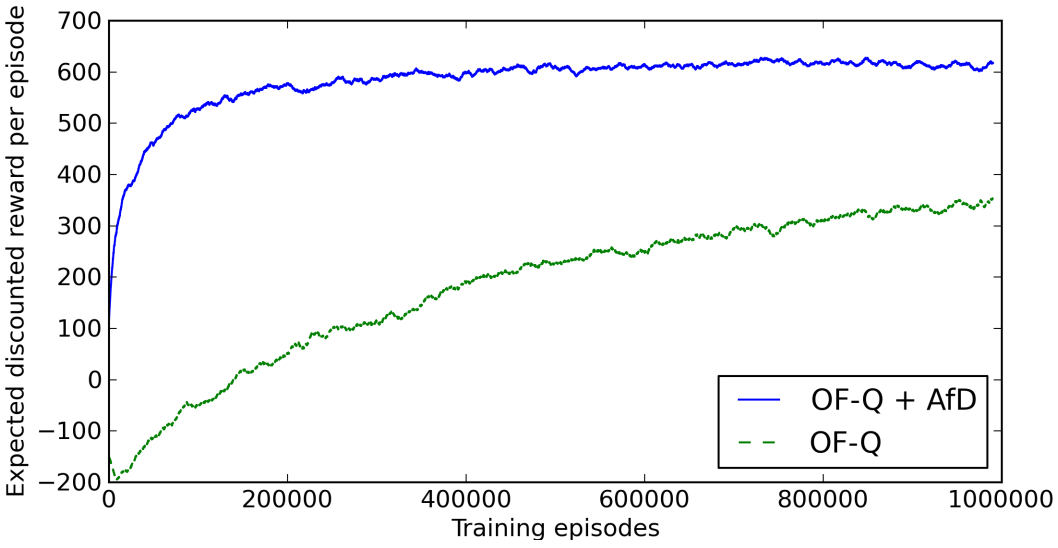


Figure 26: Results combining AfD and OF-Q, expected reward per episode averaged over 10 runs.

We acquired one hour of human demonstrations of the game, a total of 272 episodes

of which 254 where successful, containing a total of 26688 time steps. Using these demonstrations, our algorithm could identify correctly the irrelevant features for each object class (enemies and bombs). To determine the relevant features for enemies, the algorithm used the samples where only one enemy and the agent were present (there were 3661 of these samples), so that it was clear that the enemy was the main object of focus. For the bombs, there were no cases where there was only one bomb and the agent; at least one enemy was also always present (there were 453 of these samples). However, given the low number of objects, the case with one enemy and one bomb was sufficient to identify the relevant features of the bombs. Figure 26 shows that in the Space Invaders domain with irrelevant features, OF-Q combined with AfD performs significantly better than OF-Q alone.

8.4 Discussion and generalization

The application of AfD within each object class in OF-Q allows a significant speed-up, exponential in the number of features, as seen in Section 6.5. We have experimentally shown that this speed-up occurs, but we have limited ourselves to easy cases in which some of the samples from demonstrations have few objects instances, so we can assume the focus is in a specific object and use mutual information between object features and teacher actions to determine the relevant features.

A more general combination of AfD and OF-Q requires a way of identifying which object occupies the attention of the teacher at each moment. With this additional information, it would be possible to apply AfD feature selection even in complex domains in which there are a high number of objects at every time step. Further, a clear separation between samples for each specific object would allow implementation of ADA-like object-specific subtask decomposition, which useful in highly complex domains. The development of such techniques is outside the scope of this work, but we shall discuss several options for obtaining this.

The easiest option would be to implement a system for the teacher to annotate her demonstrations according to which object she was paying attention at each moment. Such a system can work well in some domains, but a generic implementation would not be easy, and the teacher would have to be trained to make the annotations. More importantly, in many tasks, the teacher may not be conscious of what object she was really paying attention to while performing the task, so the annotations might be incomplete or unreliable.

Another option to identify the focus of attention of an individual in an interactive task is an eye-tracking device [20]. Knowing which object the player is paying attention to at each moment would allow the partitioning of the demonstrations according to the class of the object that was in focus at each moment. Then, considering the state of the object in focus, the application of AfD would be immediate.

Yet another approach is to allow the teacher to stage demonstrations with simple cases (a few object instances) where it is clear which object is in focus. This idea is similar to previous work in training regimes [83]. It has been shown that non-expert teachers are able to provide effective regimens while enjoying the task.

Any of these techniques can be used to combine attention focus derived from human demonstrations and attention focus derived from a world model. Such composition combines the speed-ups of both techniques, enabling agents to learn policies for more complex domains than the use of the two techniques alone allows.

CHAPTER IX

CONCLUSIONS

We have shown that attention focus algorithms can indeed offer significant speed-ups over previous reinforcement learning algorithms with respect to the number of state features in complex domains. These speed-ups are exponential when using table-based representations. In the case of combination with state-of-the-art function approximation algorithms, such as fitted Q-learning and LSPI, the speed-ups are polynomial, allowing an agent to quickly solve problems that, without using attention focus, would have been beyond its possibilities because of space and time constraints.

We have demonstrated two different sources of focus information: demonstrations from non-expert human teachers and generic world models combined with the experience of the agent at performing the task. Regarding demonstrations, we have shown that it is possible to derive the attention focus from humans just by observing them while they try to complete the task, without needing them to do any additional work or provide any additional information. With respect to world models, we have shown that it is possible to use general assumptions about object independence to learn good policies quickly. We have also shown that, in cases where objects are not independent, it is possible to learn the dependencies between objects just from interactions with the environment. Finally, we have shown that these two sources of focus information can be combined for further speed-ups.

Our algorithms do not find optimal policies, but satisficing ones. This is a necessary trade-off for algorithms that solve MDPs that may represent a complex, *e.g.,* NP-complete, problem. In our experiments, previous algorithms that do guarantee optimality could barely move beyond the performance of a random policy even after

several orders of magnitude more iterations than our algorithms.

To conclude, emphasize that we hope our work will help shift some of the machine learning research focus from algorithms to representations. A lot of research takes a suitable representation for granted. However, this is often not the case with autonomous agents that must learn unforeseen tasks. We consider this work a small step towards solving the problem of how we can automatically derive better state-space representations. We hope our work will inspire other researchers to consider this problem in their future work.

APPENDIX A

MUTUAL INFORMATION

Mutual information is a measure of the amount of *entropy* in one random variable that can be explained by the value of a different random variable,

$$I(X;Y) = H(X) - H(X|Y) = H(X) + H(Y) - H(X,Y),$$

where $H(X)$ is the entropy of random variable X.

Mutual information can be computed as

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \left(\frac{p(x,y)}{p_x(x) p_y(y)} \right), \tag{18}$$

where $p(x,y)$ is the joint *probability density function* (pdf) of random variables X and Y, and $p_x(x)$ and $p_y(x)$ are the respective marginal pdfs of each random variable.

REFERENCES

[1] Abbeel, P. and Ng, A. Y., "Apprenticeship learning via inverse reinforcement learning," *Proc. Int. Conf. on Machine Learning*, p. 1, 2004.

[2] Aler, R., Garcia, O., and Valls, J., "Correcting and Improving Imitation Models of Humans for Robosoccer Agents," *2005 IEEE Congress on Evolutionary Computation*, pp. 2402–2409, 2005.

[3] Aloupis, G., Demaine, E., and Guo, A., "Classic Nintendo Games are (NP-) Hard," *Arxiv preprint arXiv:1203.1895*, pp. 1–21, 2012.

[4] Argall, B. D., Chernova, S., Veloso, M., and Browning, B., "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, pp. 469–483, May 2009.

[5] Azar, M., Munos, R., and Ghavamzadeh, M., "Speedy Q-Learning," *Advances in Neural Information Processing Systems*, pp. 1–9, 2011.

[6] Barto, A., "Recent advances in hierarchical reinforcement learning," *Discrete Event Dynamic Systems*, vol. 13, pp. 341–379, 2003.

[7] Beitelspacher, J., Fager, J., Henriques, G., and McGovern, A., "Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout," Tech. Rep. February, School of Computer Science, University of Oklahoma, 2006.

[8] Bradski, G., "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[9] Brafman, R. I. and Tennenholtz, M., "R-max - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning," *Journal of Machine Learning Research*, vol. 3, pp. 213–231, Feb. 2003.

[10] Busoniu, L., Lazaric, A., Ghavamzadeh, M., Munos, R., Babuska, R., and Schutter, B. D., "Least-squares methods for policy iteration," *Reinforcement Learning*, pp. 75–109, 2012.

[11] Chapman, D. and Kaelbling, L., "Input generalization in delayed reinforcement learning: An algorithm and performance comparisons," *Proc. Int. Joint Conf. on Artificial Intelligence*, pp. 726–731, 1991.

[12] Coates, A., Abbeel, P., and Ng, A., "Apprenticeship learning for helicopter control," *Communications of the ACM*, no. Icml, pp. 144–151, 2009.

[13] Cobo, L. C., Isbell, C. L., and Thomaz, A. L., "Automatic task decomposition and state abstraction from demonstration," *Proc. Int. Conf. on Autonomous Agents and Multiagent Systems*, 2012.

[14] Cobo, L. C., Isbell, C. L., and Thomaz, A. L., "Object Focused Q-learning for Autonomous Agents," *Proc. Int. Conf. on Autonomous Agents and Multiagent Systems*, 2013.

[15] Cobo, L. C., Zang, P., Isbell, C. L., and Thomaz, A. L., "Automatic state abstraction from demonstration," *Proc. Int. Joint Conf. on Artificial Intelligence*, 2011.

[16] Cowan, N., "The Magical Mystery Four: How Is Working Memory Capacity Limited, and Why?," *Current Directions in Psychological Science*, vol. 19, pp. 51–57, Feb. 2010.

[17] Dietterich, T., "The MAXQ method for hierarchical reinforcement learning," in *Proc. Int. Conf. on Machine Learning*, pp. 118–126, 1998.

[18] Diuk, C. and Cohen, A., "An object-oriented representation for efficient reinforcement learning," *Proc. Int. Conf. on Machine Learning*, 2008.

[19] Domingos, P., "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, p. 78, Oct. 2012.

[20] Duchowski, A. T., "A breadth-first survey of eye-tracking applications.," *Behavior research methods, instruments, & computers : a journal of the Psychonomic Society, Inc*, vol. 34, pp. 455–70, Nov. 2002.

[21] Dzeroski, S., Raedt, L. D., and Blockeel, H., "Relational reinforcement learning," in *Int. Conf. on Machine Learning*, 2001.

[22] Elkan, C. and Noto, K., "Learning classifiers from only positive and unlabeled data," *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data mining*, p. 213, 2008.

[23] Farahmand, A.-m., Ghavamzadeh, M., Szepesvari, C., and Mannor, S., "Regularized policy iteration," *Advances in Neural Information Processing Systems*, 2009.

[24] Fern, A., Yoon, S., and Givan, R., "Approximate policy iteration with a policy language bias," in *Advances in Neural Information Processing Systems*, 2003.

[25] Finney, S., Gardiol, N., Kaelbling, L., and Oates, T., "Learning with deictic representation," Tech. Rep. April, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2002.

[26] Gelly, S., Schoenauer, M., Sebag, M., Kocsis, L., Paris-sud, U., Cedex, O., and Silver, D., "The Grand Challenge of Computer Go : Monte Carlo Tree Search and Extensions," *Communications of the ACM*, 2012.

[27] Geramifard, A., Bowling, M., Zinkevich, M., and Sutton, R., "iLSTD : Eligibility Traces and Convergence Analysis," in *Advances in Neural Information Processing Systems*, p. 441, The MIT Press, 2007.

[28] Greiner, R., "PALO: A probabilistic hill-climbing algorithm," *Artificial Intelligence*, 1995.

[29] Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N., "Generalizing Plans to New Environments in Relational MDPs," *International Joint Conference on Artificial Intelligence*, no. August, 2003.

[30] Hall, M., *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[31] Hengst, B., "Discovering hierarchy in reinforcement learning with HEXQ," in *Proc. Int. Conf. on Machine Learning*, pp. 234–250, 2002.

[32] Jong, N. K. and Stone, P., "State Abstraction Discovery from Irrelevant State Variables," *Proc. Int. Joint Conf. on Artificial Intelligence*, no. August, pp. 752–757, 2005.

[33] Jong, N., Hester, T., and Stone, P., "The utility of temporal abstraction in reinforcement learning," *Proc. Int. Conf. on Autonomous Agents and Multiagent Systems*, pp. 299–306, 2008.

[34] Jonsson, A. and Barto, A., "Automated state abstraction for options using the U-tree algorithm," *Advances in Neural Information Processing Systems*, pp. 1054–1060, 2001.

[35] Kearns, M. and Singh, S., "Finite-Sample Convergence Rates for Q-Learning and Indirect Algorithms," in *Advances in Neural Information Processing Systems*, pp. 996–1002, 1999.

[36] Keller, P. W., Mannor, S., and Precup, D., "Automatic basis function construction for approximate dynamic programming and reinforcement learning," *Proc. Int. Conf. on Machine Learning*, pp. 449–456, 2006.

[37] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by simulated annealing.," *Science (New York, N.Y.)*, vol. 220, pp. 671–80, May 1983.

[38] Knox, W. and Stone, P., "Combining Manual Feedback with Subsequent MDP Reward Signals for Reinforcement Learning," *Proc. Int. Conf. on Autonomous Agents and Multiagent Systems*, pp. 10–14, 2010.

[39] Kolter, J. Z. and Ng, A. Y., "Regularization and feature selection in least-squares temporal difference learning," *Proc. Int. Conf. on Machine Learning*, vol. 94305, pp. 1–8, 2009.

[40] Kolter, J. Z. and Ng, A. Y., "Regularization and feature selection in least-squares temporal difference learning," in *Proc. Int. Conf. on Machine Learning*, pp. 521–528, 2009.

[41] Konidaris, G. and Barto, A., "Efficient skill learning using abstraction selection," in *Proc. Int. Joint Conf. on Artificial Intelligence*, pp. 1107–1112, 2009.

[42] Konidaris, G., Niekum, S., and Thomas, P. S., "TD γ : Re-evaluating Complex Backups in Temporal Difference Learning," in *Advances in Neural Information Processing Systems*, pp. 1–9, 2011.

[43] Lagoudakis, M. G. and Parr, R., "Least-Squares Policy Iteration," *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, Jan. 2003.

[44] Lee, H., Grosse, R., Ranganath, R., and Ng, A., "Unsupervised learning of hierarchical representations with convolutional deep belief networks," *Communications of the ACM*, vol. 54, no. 10, pp. 95–103, 2011.

[45] Letouzey, F., Denis, F., and Gilleron, R., "Learning from positive and unlabeled examples," in *Algorithmic Learning Theory*, pp. 71–85, Springer, 2009.

[46] Li, L., "A worst-case comparison between temporal difference and residual gradient with linear function approximation," in *Proc. Int. Conf. on Machine Learning*, 2008.

[47] Li, L. and Littman, M. L., "Reducing reinforcement learning to KWIK online regression," *Annals of Mathematics and Artificial Intelligence*, vol. 58, pp. 217–237, June 2010.

[48] Li, L., Walsh, T., and Littman, M., "Towards a unified theory of state abstraction for MDPs," in *Proc. Int. Symp. on Artificial Intelligence and Mathematics*, pp. 531–539, Citeseer, 2006.

[49] Maei, H., Szepesvári, C., Bhatnagar, S., and Sutton, R., "Toward off-policy learning control with function approximation," in *Proc. Int. Conf. on Machine Learning*, pp. 719–726, 2010.

[50] Mannor, S., Menache, I., Hoze, A., and Klein, U., "Dynamic abstraction in reinforcement learning via clustering," *Proc. Int. Conf. on Machine Learning*, p. 71, 2004.

[51] McCallum, A., "Learning to use selective attention and short-term memory in sequential tasks," *From animals to animats*, vol. 4, pp. 315–324, 1996.

[52] McGovern, A. and Barto, A., "Automatic discovery of subgoals in reinforcement learning using diverse density," in *Proc. Int. Conf. on Machine Learning*, pp. 361–368, Citeseer, 2001.

[53] Mehta, N., Wynkoop, M., Ray, S., Tadepalli, P., and Dietterich, T., "Automatic induction of MAXQ hierarchies," in *NIPS Workshop: Hierarchical Organization of Behavior*, pp. 1–5, Citeseer, 2007.

[54] Minut, S. and Mahadevan, S., "A reinforcement learning model of selective visual attention," in *Proc. Int Conf. on Autonomous Agents*, 2001.

[55] Ng, A. Y., Kim, H. J., Jordan, M. I., and Sastry, S., "Autonomous helicopter flight via Reinforcement Learning," in *Advances in Neural Information Processing Systems*, vol. 16, 2004.

[56] Ng, A. and Jordan, M., "PEGASUS: A policy search method for large MDPs and POMDPs," in *Proc. of the Conf. on Uncertainty in Artificial Intelligence*, pp. 406–415, Citeseer, 2000.

[57] Nø rretranders, T., *The user illusion: Cutting consciousness down to size.* Viking, 1991.

[58] Otterlo, M. V., "A Survey of Reinforcement Learning in Relational Domains," tech. rep., Centre for Telematics and Information Technology University of Twente, 2005.

[59] Parr, R., Painter-Wakefield, C., Li, L., and Littman, M., "Analyzing feature generation for value-function approximation," *Proc. Int. Conf. on Machine Learning*, pp. 737–744, 2007.

[60] Precup, D. and Sutton, R., "Off-policy temporal-difference learning with function approximation," in *Proc. Int. Conf. on Machine Learning*, 2001.

[61] Quinlan, J., *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

[62] Rao, K. and Whiteson, S., "V-MAX: tempered optimism for better PAC reinforcement learning," *Proc. Int. Conf. on Autonomous Agents and Multiagent Systems*, pp. 375–382, 2012.

[63] Riedmiller, M., "Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method," in *Proc. European Conf. on Machine Learning*, pp. 317–328, 2005.

[64] Russell, S. and Zimdars, A., "Q-decomposition for reinforcement learning agents," in *Proc. Int. Conf. on Machine Learning*, 2003.

[65] Seijen, H. V., Whiteson, S., and Kester, L., "Switching between representations in reinforcement learning," *Interactive Collaborative Information Systems*, pp. 65–84, 2010.

[66] Simon, H. A., *Administrative behavior*, vol. 4. Cambridge University Press, 1957.

[67] Singh, S., Jaakkola, T., and Jordan, M., "Learning without state-estimation in partially observable Markovian decision processes," *Proc. Int. Conf. on Machine Learning*, pp. 284–292, 1994.

[68] Smart, W. and Pack Kaelbling, L., "Effective reinforcement learning for mobile robots," *Proceedings 2002 IEEE International Conference on Robotics and Automation*, pp. 3404–3410, 2002.

[69] Sprague, N. and Ballard, D., "Multiple-goal reinforcement learning with modular sarsa(0)," *Proc. Int. Joint Conf. on Artificial Intelligence*, 2003.

[70] Stolle, M. and Precup, D., "Learning Options in Reinforcement Learning," *Abstraction, Reformulation, and Approximation*, pp. 212–223, 2002.

[71] Sutton, R. S., Precup, D., and Singh, S., "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, pp. 181–211, Aug. 1999.

[72] Sutton, R. and Barto, A., *Reinforcement learning: An introduction*. Cambridge University Press, 1998.

[73] Sutton, R., Maei, H., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, C., and Wiewiora, E., "Fast gradient-descent methods for temporal-difference learning with linear function approximation," in *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 993–1000, ACM, 2009.

[74] Sutton, R., McAllester, D., and Singh, S., "Policy gradient methods for reinforcement learning with function approximation," *Advances in Neural Information Processing Systems*, pp. 1057–1063, 2000.

[75] Szepesvari, C., "Algorithms for Reinforcement Learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, pp. 1–98, 2010.

[76] Talvitie, E. and Singh, S., "Simple Local Models for Complex Dynamical Systems," in *Advances in Neural Information Processing Systems*, 2008.

[77] Tesauro, G., "Programming backgammon using self-teaching neural nets," *Artificial Intelligence*, vol. 134, pp. 181–199, Jan. 2002.

[78] Vigorito, C. M. and Barto, A. G., "Intrinsically Motivated Hierarchical Skill Learning in Structured Environments," *IEEE Trans. on Autonomous Mental Development*, vol. 2, no. 2, pp. 132–143, 2010.

[79] Watkins, C. J. C. H. and Dayan, P., "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[80] Whiteson, S. and Stone, P., "Evolutionary function approximation for reinforcement learning," *The Journal of Machine Learning Research*, vol. 7, pp. 877–917, 2006.

[81] Wilson, T., *Strangers to ourselves: Discovering the adaptive unconscious*. Harvard University Press, 2002.

[82] Xu, X., Hu, D., and Lu, X., "Kernel-based least squares policy iteration for reinforcement learning.," *IEEE Trans. on Neural Networks*, vol. 18, pp. 973–92, July 2007.

[83] Zang, P., Irani, A., Zhou, P., Jr, C. I., and Thomaz, A., "Using Training Regimens to Teach Expanding Function Approximators," in *Proceedings of the Ninth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2010.

[84] Zang, P., Zhou, P., Minnen, D., and Isbell, C., "Discovering options from example trajectories," *Proc. Int. Conf. on Machine Learning*, pp. 1217–1224, 2009.

[85] Zhang, M. and Zhou, Z., "ML-KNN: A lazy learning approach to multi-label learning," *Pattern Recognition*, vol. 40, pp. 2038–2048, July 2007.

[86] imek, O. and Barto, A. G., "Using relative novelty to identify useful temporal abstractions in reinforcement learning," *Proc. Int. Conf. on Machine Learning*, p. 95, 2004.

VITA

Luis Carlos Cobo Rus (Luis C. Cobo) was born in Úbeda, Spain in 1983. In 2006, he received a double degree of *Ingeniero Superior de Telecomunicación* by the *Universidad Politécnica de Madrid* and M.Sc. in Electrical Engineering by the Illinois Institute of Technology. From 2006 to 2008, he worked in the San Francisco-based cozybit Inc. on the wireless mesh stacks (802.11s) of the One Laptop per Child project and the Linux kernel. He graduated with a Ph.D. in Electrical and Computer Engineering at the Georgia Institute of Technology in 2013. His research interests include reinforcement learning, learning from demonstration, automatic feature engineering and socially guided machine learning.